



# Fault Localization to Detect Co-change Fixing Locations

Yi Li

New Jersey Institute of Technology

New Jersey, USA

yl622@njit.edu

Shaohua Wang\*

New Jersey Institute of Technology

New Jersey, USA

davidsw@njit.edu

Tien N. Nguyen

University of Texas at Dallas

Texas, USA

tien.n.nguyen@utdallas.edu

## ABSTRACT

Fault Localization (FL) is a precursor step to most Automated Program Repair (APR) approaches, which fix the faulty statements identified by the FL tools. We present **FixLOCATOR**, a Deep Learning (DL)-based fault localization approach supporting the detection of faulty statements in one or multiple methods that need to be modified accordingly in the same fix. Let us call them *co-change (CC) fixing locations* for a fault. We treat this FL problem as dual-task learning with two models. The method-level FL model, `MethFL`, learns the methods to be fixed together. The statement-level FL model, `stmtFL`, learns the statements to be co-fixed. Correct learning in one model can benefit the other and vice versa. Thus, we simultaneously train them with soft-sharing the models' parameters via cross-stitch units to enable the propagation of the impact of `MethFL` and `stmtFL` onto each other. Moreover, we explore a novel feature for FL: the co-changed statements. We also use Graph-based Convolution Network to integrate different types of program dependencies.

Our empirical results show that FixLOCATOR relatively improves over the state-of-the-art statement-level FL baselines by locating 26.5%–155.6% more CC fixing statements. To evaluate its usefulness in APR, we used FixLOCATOR in combination with the state-of-the-art APR tools. The results show that FixLOCATOR+DEAR (the original FL in DEAR replaced by FixLOCATOR) and FixLOCATOR+CURE improve relatively over the original DEAR and Ochiai+CURE by 10.5% and 42.9% in terms of the number of fixed bugs.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Fault Localization; Deep Learning; Co-Change Fixing Locations

### ACM Reference Format:

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. Fault Localization to Detect Co-change Fixing Locations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3540250.3549137>

\*Corresponding Author

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 ACM 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.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9413-0/22/11...\$15.00

<https://doi.org/10.1145/3540250.3549137>

## 1 INTRODUCTION

To assist developers in the bug-detecting and fixing process, several approaches have been proposed for *Automated Program Repair* (APR) [21]. A common usage of an APR tool is that one needs to use a *fault localization* (FL) tool [41] to locate the faulty statements that must be fixed, and then uses an APR tool to generate the fixing changes for those detected statements. The input of an FL model is the execution of a test suite, in which some of the test cases are passing or failing ones. Specifically, the key input is the *code coverage matrix* in which the rows and columns correspond to the statements and test cases, respectively. Each cell is assigned with the value of 1 if the statement is executed in the respective test case, and with the value of 0, otherwise. An FL model uses such information to identify the list of *suspicious lines of code* that are ranked based on their associated *suspiciousness scores* [41]. In recent advanced FL, several approaches also support fault localization at method level to locate faulty methods [22, 24].

The FL approaches can be broadly divided into the following categories: *spectrum-based fault localization* (SBFL) [6, 17, 18], *mutation-based fault localization* (MBFL) [31, 34, 35], and *machine learning* (ML) and *deep learning* (DL) *fault localization* [22, 24]. For SBFL approaches, the key idea is that a line covered more in the failing test cases than in the passing ones is more suspicious than a line executed more in the passing ones. To improve SBFL, MBFL approaches [31, 34, 35] enhance the code coverage matrix by modifying a statement with mutation operators, and collecting code coverage when executing the mutated programs with the test cases. The MBFL approaches apply suspiciousness score formulas in the same manner as in SBFL approaches on the matrix for each original statement and its mutated code. Finally, ML and DL-based FL approaches explore the code coverage matrix and apply different neural network models for fault localization.

Despite their successes, the state-of-the-art FL approaches are still limited in locating all dependent fixing locations that need to be repaired at the same time in the same fix. In practice, there are many bugs that require *dependent changes in the same fix to multiple lines of code in one or multiple hunks of the same or different methods for the program to pass the test cases*. For those bugs, applying the fixing change to individual statements once at a time will not make the program pass the test case after the change to one statement. This capability to detect the fixing locations of the co-changes in a fix for a bug (let us call them *Co-change (CC) Fixing Locations*) is crucial for an APR tool. Such capability will enable an APR tool to *make the correct and complete changes to fix a bug*.

The state-of-the-art FL approaches do not satisfy that requirement. From the ranked list of suspicious statements returned from an existing FL model, a naive approach to detect CC fixing locations would be to take the top  $k$  statements in that list and to consider them as to be fixed together. This solution might be ineffective

because the mechanisms used in the state-of-the-art FL approaches have never considered the co-change nature of those fixes. Our empirical evaluation also confirmed that (Section 8.1).

Detecting all the CC fixing locations at multiple statements in potentially multiple methods is challenging. A naive solution would be detecting the potential methods that need to be fixed together and then detecting potential statements that need to be changed together in each of those methods. However, doing so will create a confounding effect from the inaccuracy of the detection of the co-fixed methods to that of the co-fixed statements.

We propose FixLOCATOR, a fault localization approach to derive the co-change fixing locations in the same fix for a fault (i.e., multiple faulty statements in possible multiple faulty methods). To avoid the confounding effect in that naive solution, we treat this problem as *dual-task learning* with two dedicated models. First, the *method-level FL* model (`MethFL`) learns the methods that need to be modified in the same fix. Second, the *statement-level FL* model (`StmtFL`) learns the co-fixed statements in the same or different methods. The intuition is that they are closely related, which we refer to as *duality*. *Correct learning for a model can benefit the other and vice versa*. If two statements in two methods are fixed together for a bug, those methods are also co-fixed. If two methods are co-fixed, some of their statements are also co-fixed. Exploring this duality can provide useful constraints to detect CC fixing locations for a bug. Thus, instead of cascading the two models `MethFL` and `StmtFL`, we train them simultaneously with the soft-sharing of the models' parameters to exploit this duality. Specifically, we leverage the cross-stitch units [30] to connect `MethFL` and `StmtFL`. In a cross-stitch unit, the sharing of representations between `MethFL` and `StmtFL` is modeled by learning a linear combination of the input features from the two models. The cross-stitch units enable the *propagation of the impact of MethFL and StmtFL on each other*.

In addition to the new solution in dual-task learning, we utilize a novel feature for this CC fixing location problem: *co-changed statements*, which have never been exploited in FL. The rationale is that the co-changed statements in the past might become the statements that will be fixed together in the future. Finally, since the co-fixed statements are often interdependent, we use Graph-based Convolution Network (GCN) [26] to integrate different types of program dependencies among statements, e.g., data and control dependencies, execution traces, stack traces, etc. We also encode test coverage and co-changed/co-fixed statements in the graph. The GCN model learns and predicts the bugginess of the statements.

We conducted several experiments to evaluate FixLOCATOR on Defects4J-v2.0 [1]. Our empirical results show that FixLOCATOR improves the baselines, CNN-FL [50], DeepFL [22], DeepRL4FL [24], and DEAR's FL [25] by 16.6%, 16.9%, 9.9%, and 20.6% respectively, in terms of Hit-1 (i.e., the percentage of bugs in which the predicted set overlaps with the oracle set for *at least* one faulty statement), and by 33.6%, 40.3%, 26.5%, and 57.5% in terms of Hit-2 (i.e., the percentage of bugs in which the number of overlapping statements between the predicted and oracle sets is  $\geq 2$ ), 43.9%, 46.4%, 28.1%, and 51.9% in terms of Hit-3, respectively. FixLOCATOR also improves those baselines by 32.0%, 38.8%, 20.8%, and 46.1% in terms of Hit-All (i.e., the predicted set exactly matches with the oracle set for a bug).

To evaluate its usefulness in APR, we combined it with the APR tools, DEAR [25] and CURE [15]. We replaced DEAR's FL module

```

1 public void toSource(final CodeBuilder cb, int inputSeqNum, Node root) {
2     ...
3     - String code = toSource(root, sourceMap);
4     + String code = toSource(root, sourceMap, inputSeqNum == 0);
5     if (!code.isEmpty()) {
6         cb.append(code);
7     } ...
8 }
9 /**
10 * @Override
11 String toSource(Node n) {
12     initCompilerOptionsIfTesting();
13     - return toSource(n, null);
14     + return toSource(n, null, true);
15 }
16 /**
17 - private String toSource(Node n, SourceMap sourceMap)
18 + private String toSource(Node n, SourceMap sourceMap, boolean firstOutput)
19     .....
20     builder.setSourceMapDetailLevel(options.sourceMapDetailLevel);
21     - builder.setTagAsStrict(
22     + builder.setTagAsStrict(firstOutput &&
23         options.getLanguageOut(a) == LanguageMode.ECMASCIPTS_STRICT);
24     builder.setLineLengthThreshold(options.lineLengthThreshold);
25     .....
26 }
```

**Figure 1: Co-Change Fixing Locations for a Fault**

with FixLOCATOR for a variant, DEAR<sup>FixL</sup>. Our result shows that DEAR<sup>FixL</sup> and FixLOCATOR+CURE improve relatively DEAR and Ochiai+CURE by 10.5% and 42.9% in terms of numbers of fixed bugs.

Through our ablation analysis on the impact of different features and modules of FixLOCATOR, we showed that all designed features/modules have contributed to its high performance. Specifically, the proposed dual-task learning significantly improves the statement-level FL by up to 12.8% in terms of Hit-1. The designed feature of co-change relations among methods and statements has also positively contributed to FixLOCATOR's high accuracy level.

The contributions of this paper are listed as follows:

- (1) **FixLOCATOR: Advancing DL-based Fault Localization** to derive the **co-change fixing locations** (multiple faulty statements) in the same fix for a bug. We treat that problem as *dual-task learning to propagate the impact* between the method-level and statement-level FL.
- (2) **Novel graph-based representation learning with GCN and novel type of features in co-changed statements for FL** enable dual-task learning to derive CC fixing locations.
- (3) **Extensive empirical evaluation.** We evaluated FixLOCATOR against the recent DL-based FL models to show its accuracy and usefulness in APR. Our data/tool are available [3].

## 2 MOTIVATING EXAMPLE

### 2.1 Example and Observations

Let us start with a real-world example. Figure 1 shows a bug fix in the Defects4J dataset that require multiple interdependent changes to multiple statements in different methods. The bug occurred when the method call to `setTagAsStrict` did not consider the first output in its arguments. Therefore, for fixing, a developer adds a new argument in the method `toSource` at line 18, and uses that argument in the method call `setTagAsStrict (firstOutput, ...)` at line 22. Because the method `toSource` at line 17 was changed, the two callers at line 3 of the method `toSource` (line 1) and at line 13 of the method `toSource` (line 11) need to be changed accordingly.

**2.1.1 Observation 1 [Co-change Fixing Locations].** In this example, the changes to fix this bug involve multiple faulty statements that are dependent on one another. Fixing only one of the faulty statements will not make the program pass the failing test(s). Fixing individual statements once at a time in the ranked list returned from an existing FL tool will also not make the program pass the tests. For an APR model to work, an FL tool needs to point out all of those faulty statements to be changed in the same fix. For example, all four faulty statements at lines 17, 21, 3, and 13 need to be modified accordingly in the same fix to fix the bug in Figure 1.

**2.1.2 Observation 2 [Multiple Faulty Methods].** As seen, this bug requires an APR tool to make changes to multiple statements in three different methods in the same fix: `toSource(...)` at lines 17 and 21, `toSource(...)` at line 3, and `toSource(...)` at line 13. Thus, it is important for an FL tool to connect and identify these multiple faulty statements in potentially different methods.

Traditional FL approaches [49, 52] using program analysis (PA), e.g., execution flow analysis, are restricted to specific PA techniques, thus, not general to locate all types of CC fixing locations. Spectrum-based [6, 16], mutation-based [31, 34, 35], statistic-based [27], and machine learning (ML)-based FL approaches [22, 24] could implicitly learn the program dependencies for FL purpose. However, despite their successes, the non-PA FL approaches *do not support the detection of multiple locations that need to be changed in the same fix for a bug, i.e., Co-Change (CC) Fixing Locations*. The spectrum-based and ML-based FL models return a ranked list of suspicious statements according to the corresponding suspiciousness scores. In this example, the lines 13, 17, 21, and the other lines (e.g., 12, 20 and 24) are executed in the same passing or failing test cases, thus assigned with the same scores by spectrum- and mutation-based FL approaches. A user would not be informed on what lines need to be fixed together. Those non-PA, especially ML-based FL approaches, do not have a mechanism to detect CC fixing locations.

In this work, we aim to advance the level of *deep learning (DL)-based FL approaches to detect CC fixing statements*. However, it is not trivial. A solution of assuming the top- $k$  suspicious statements from a FL tool as CC fixing locations does not work because even being the most suspicious, those statements might not need to be changed in the same fix. In this example, all of the above lines with the same suspiciousness scores would confuse a fixer.

Moreover, another naive solution would be to use a method-level FL tool to detect multiple faulty methods first and then use a statement-level FL tool to detect the statements within each faulty method. As we will show in Section 7, the inaccuracy of the first phase of detecting faulty methods will have a confounding effect on the overall performance in detecting CC fixing statements.

## 2.2 Key Ideas

We propose FixLOCATOR, an FL approach to locate all the CC fixing locations (i.e., faulty statements) that need to be changed in the same fix for a bug. In designing FixLOCATOR, we have the following key ideas in both new model and new features:

**2.2.1 Key Idea 1 [Dual-Task Learning for Fault Localization].** To avoid the confounding effect in a naive solution of detecting faulty methods first and then detecting faulty statements in those methods,

we design an approach that treats this FL problem of detecting dependent CC fixing locations as *dual-task learning* between the method-level and statement-level FL. First, the *method-level FL* model (`MethFL`) aims to learn the methods that need to be modified in the same fix. Second, the *statement-level FL* model (`StmtFL`) aims to learn the co-fixing statements regardless of whether they are in the same or different methods.

Intuitively, `MethFL` and `StmtFL` are related to each other, in which the results of one model can help the other. We refer to this relation as *duality*, which can provide some useful constraints for FixLOCATOR to learn dependent CC fixing locations. We conjecture that the joint training of the two models can improve the performance of both models, when we leverage the constraints of this duality in term of shared representations. For example, if two statements in two different methods  $m_1$  and  $m_2$  were observed to be changed in the same fix, then it should help the model learn that  $m_1$  and  $m_2$  were also changed together to fix the bug. If two methods were observed to be fixed together, then some of their statements were changed in the same fix as well. In our model, we jointly train `MethFL` and `StmtFL` with the soft-sharing of the models' parameters to exploit their relation. Specifically, we use a mechanism, called *cross-stitch unit* [30], to learn a linear combination of the input features from those two models to *enable the propagation of the impact of MethFL and StmtFL on each other*. We also add an attention mechanism in the two models to help emphasize on the key features.

**2.2.2 Key Idea 2 [Co-change Representation Learning in Fault Localization].** In detecting CC fixing locations, in addition to a new dual-task learning model in key idea 1, we use a new feature: *co-change information* among statements/methods, which has never explored in prior fault localization research. The rationale is that the co-changed statements/methods in the past might become the statements/methods that will be fixed together for a bug in the future. We also encode the co-fixed statements/methods in the same fixes. The co-changed/co-fixed statements/methods in the same commit are used to train the models.

**2.2.3 Key Idea 3 [Graph Modeling for Dependencies among Statements/Methods].** The statements/methods that need to be fixed together are interdependent via several dependencies. Thus, we use Graph-based Convolution Network (GCN) [26] to model different types of dependencies among statements/methods, e.g., data and control dependencies in a program dependence graph (PDG), execution traces, stack traces, etc. We encode the *co-change/co-fix relations* into the graph representations with different types of edges representing different relations. The GCN model enables nodes' and edges' attributes and learns to classify the nodes as buggy or not.

## 3 FIXLOCATOR: APPROACH OVERVIEW

### 3.1 Training Process

Figure 2 summarizes the training process. The input of training includes the passing and failing test cases, and the source code under study. The output includes the trained method-level FL model (detecting co-fixed methods) and the trained statement-level FL model (detecting co-fixed statements). The training process has three main steps: 1) *Feature Extraction*, 2) *Graph-based Feature Representation Learning*, and 3) *Dual-Task Learning Fault Localization*.

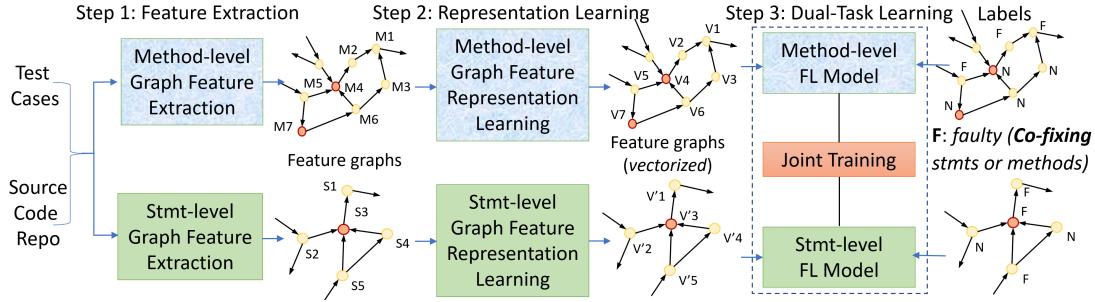


Figure 2: FIXLOCATOR: Training Process

**3.1.1 Step 1. Feature Extraction.** (Section 4). We aim to extract the important features for FL from the test coverage and source code including co-changes. The features are extracted from two levels: statements and methods. At each level, we extract the important *attributes* of statements/methods, as well as the crucial *relations* among them. We use graphs to model those attributes and relations.

For a *method m*, we collect as its attributes 1) method content: the sequences of the sub-tokens of its code tokens (excluding separators and special tokens), and 2) method structure: the Abstract Syntax Tree (AST) of the method. For the relations among methods, we extract the relations involving in the following:

- 1) *Execution flow* (the calling relation, i.e., *m* calls *n*),
- 2) *Stack trace* after a crash, i.e., the order relation among the methods in the stack trace (the dynamic information in execution and stack traces have been showed to be useful in FL [22, 24]),
- 3) *Co-change relation* in the project history (two methods that were changed in the same commit are considered to have the co-change relation),
- 4) *Co-fixing relation* among the methods (two methods that were fixed for the same bug are considered to have the co-fixing relation),

5) *Similarity*: we also extract the similar methods in the project that have been buggy before in the project history. We keep only the most similar method for each method.

For a *statement s*, we extract both static and dynamic information. First, for static data, we extract the AST subtree that corresponds to *s* to represent its structure. We also extract the list of variables in *s* together with their types, forming a sequence of names and types, e.g., “name String price int ...”. Second, for dynamic data, we encode the test coverage matrix for *s* into the feature vectors.

At both method and statement levels, we use graphs to represent the methods and statements, and their relations. Let us call them the method-level and statement-level *feature graphs*.

**3.1.2 Step 2. Graph-based Feature Representation Learning.** This step is aimed to learn the vector representations (i.e., embeddings) for the nodes in the feature graphs from step 1. The input includes the method-level and statement-level feature graphs. The output includes the embeddings for the nodes in the method-/statement-level feature graphs. The graph structures for both feature graphs are un-changed after this step.

For the content of a method or statement, we use the embedding techniques accordingly to feature representations (Section 5). For the method’s content and a list of variables in a statement, the representation is a sequence of sub-tokens. We use GloVe [36] to produce

the embeddings for all sub-tokens as we consider a method or statement as a sentence in each case. We then use Gated Recurrent Unit (GRU) [14] to produce the vector for the entire sequence.

For the structure of a method or statement, the representation is a (sub)tree in the AST. For this, we first use GloVe [36] to produce the embeddings for all the nodes in the sub-tree, considering the entire method or statement as a sentence in each case. After obtaining the sub-tree where the nodes are replaced by their GloVe’s vectors, we use TreeCaps [12], which captures well the tree structure, to produce the embedding for the entire sub-tree.

For the code coverage representation, we directly use the two vectors for coverage and passing/failing and concatenate them to produce the embedding. The embedding for the most similar buggy method is computed in the same manner as explained with GloVe and TreeCaps. Finally, the embeddings for the attributes of the nodes are used in the fully connected layers to produce the embedding for each node in the feature graph at the method level. Similarly, we obtain the feature graph at the statement level in which each node is the resulting vector of the fully connected layers.

**3.1.3 Step 3. Dual-Task Learning Fault Localization.** After the feature representation learning step, we obtain two feature graphs at the method and statement levels, in which a node in either graph is a vector representation. The two graphs are used as the input for dual-task learning. For dual-task learning, we use two Graph-based Convolution Network (GCN) models [19] for the method-level FL model (*MethFL*) and the statement-level FL model (*StmtFL*) to learn the CC fixing methods and CC fixing statements, respectively. During training, the two feature graphs at the method and statement levels are used as the inputs of *MethFL* and *StmtFL*. The two GCN models play the role of binary classifiers for the bugginess for the nodes (i.e., methods/statements). We train the two models simultaneously with soft-sharing of parameters. Details will be given in Section 6.

## 3.2 Predicting Process

The input of the prediction process (Figure 3) includes the test cases and the source code in the project. The steps 1–2 of the process is the same as in training. In step 3, the feature graph  $g_M$  at the statement level built from the source code is used as the input of the *trained StmtFL* model, which predicts the labels of the nodes in that graph. The labels indicate the bugginess of the corresponding statements in the source code, which represent the CC fixing statements. If one aims to predict the faulty methods, the trained *MethFL* model can be used on the feature graph to produce the CC fixing methods.

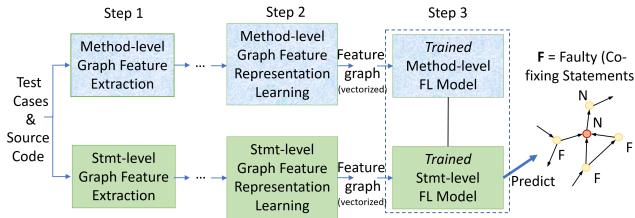
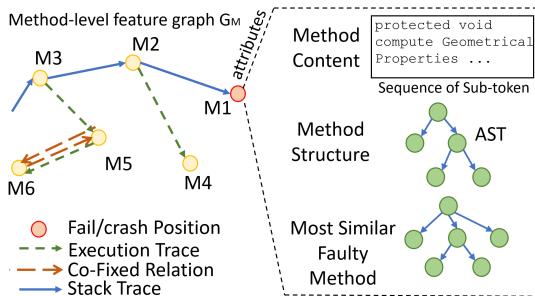


Figure 3: FIXLOCATOR: Prediction Process

Figure 4: Method-level Feature Extraction  $G_M$  for  $M_1$ 

## 4 FEATURE EXTRACTION

### 4.1 Method-Level Feature Extraction $G_M$

Figure 4 illustrates the key attributes and relations that we collect. For each method  $M_1$ , we extract the following attributes:

1) *The method's content*: we remove special characters and separators in the method's interface and body, and use naming convention to break each code token into the sub-tokens. For example, in Figure 4, the node  $M_1$  represents the method `computeGeometricalProperties` in Figure 5. For the content for  $M_1$ , the extracted sequence of sub-tokens is `protected void compute Geometrical Properties ...`

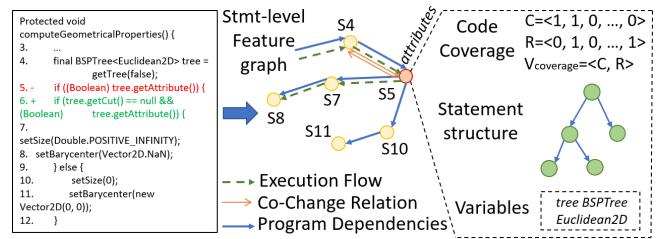
2) *The method's structure*: the corresponding parser is used to build the AST of the method (e.g., JDT [4] for Java code).

3) *Most similar faulty method*: we keep the most similar faulty method  $M_b$  with  $M_1$ . Note that we keep  $M_b$  as an attribute of  $M_1$ , rather than representing  $M_b$  in the feature graph. The rationale is that  $M_b$  might be in the past and might not be present in the current version of the project. Two methods are similar when they have similar sequences (measured by the cosine similarity) of the sub-tokens (represented by the GloVe embeddings [36]). For  $M_b$ , we build its AST and keep it as an attribute for  $M_1$ .

We encode as the edges three types of relations:

1) *Calling relation in a stack trace*: we encode into the feature graph the calling relations in a stack trace of a failed test case as we ran it. In Figure 4, a blue edge connects  $M_i$  to  $M_j$  for that relation. Since the stack trace can be long, from the failing/crash point, we collect only part of the stack trace with  $n$  levels of depth from that point. Following a prior work [44], in our experiment,  $n=10$ .

2) *Calling relations in an execution trace*: Similar to the stack trace, an execution trace needs to be encoded in the feature graph. It can be very long from the failing/crash point. Thus, we keep the methods with only  $m$  levels of length in calling relations from that point. In our experiment, we use  $m=10$ . Figure 4 illustrates a few calling relations (in green color) in execution traces.

Figure 5: Stmt-level Feature Extraction  $g_M$  for  $M_1$  in Figure 4

3) *Co-change/co-fixing relation*: Such a relation exists between two methods that were changed/fixed in a commit. Such an edge is made into two one-directional edges (e.g.,  $M_5 \xrightarrow{\text{C}} M_6$  in Figure 4).

### 4.2 Statement-Level Feature Extraction $g_M$

For each statement, we extract the following attributes:

1) *Code coverage*: we run the test cases and collect code coverage information. For each statement  $s$ , we use a vector  $C = \langle c_1, c_2, \dots, c_K \rangle$  ( $K$  is the number of test cases) to encode code coverage in which  $c_i = 1$  if the test  $t_i$  covers  $s$ , and  $c_i = 0$  otherwise. We use another vector  $R = \langle r_1, r_2, \dots, r_K \rangle$  to encode the passing/failing of a test case in which  $r_i = 1$  if the test case  $t_i$  is a passing one and  $r_i = 0$  otherwise.  $R$  is common for all the statements. We concatenate  $C$  and  $R$  for each statement to obtain the code coverage feature vector  $V_{Cov} = \langle c_1, c_2, \dots, c_K, r_1, r_2, \dots, r_K \rangle$ . We used DeepRL4FL's test ordering algorithm [24] as the ordering of test cases is useful in FL. For the different numbers of test cases across files, we perform zero padding to make the vectors have the same length.

2) *AST structure*: we extract the sub-tree in the AST that corresponds to the current statement.

3) *List of variables*: We break the names into sub-tokens. In Figure 5, the sequence for the variable list is `[tree, BSPTree, Euclidean2D, ...]`.

We encode the following types of relations among statements:

1) *Program dependence graph (PDG)*: as suggested in [24], the relations among statements in an PDG are important in FL, thus, we integrate them into the feature graph. In Figure 5, the blue edges represent the relations in the PDG for the given code. The statement at line 4 has a control/data dependency with the one at line 5, which connects to the ones at lines 7–8, and to the ones at lines 10–11.

2) *Execution flow in an execution trace*: if two statements are executed consecutively in an execution trace, we will connect them together. In Figure 5, we have the execution flow  $S_5 \rightarrow S_7, S_7 \rightarrow S_8$ .

3) *Co-change/co-fixing relation*: we maintain the co-change/co-fixing relations among statements. In Figure 5,  $S_4$  and  $S_5$  have been changed in a commit, thus, two co-change edges connect them.

## 5 FEATURE REPRESENTATION LEARNING

The goal of this step is to learn to build the vector representations for the nodes in the feature graphs at the method and statement levels. At either level, the input includes the attributes of either a method or a statement as in Figures 4 and 5. The output is each feature graph in which the nodes are replaced by their embeddings.

### 5.1 Method-Level Representation Learning

Figure 6 shows how we build the vectors for a method's attributes.

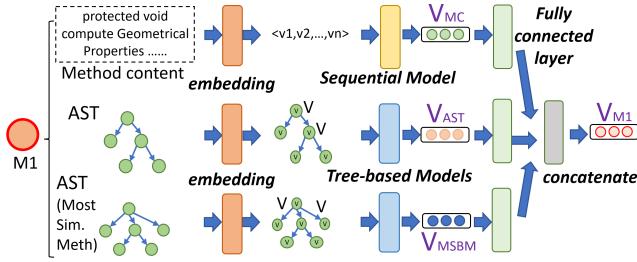


Figure 6: Method-level Feature Representation Learning

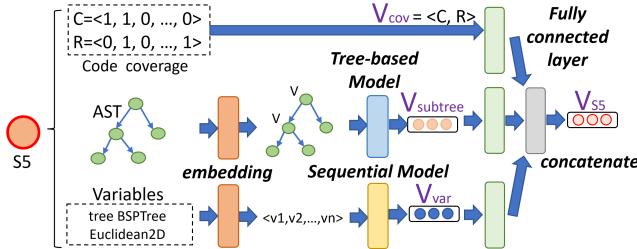


Figure 7: Statement-level Feature Representation Learning

1) *The method’s content*: the method’s content is represented by the sequence  $Seq_c$  of the sub-tokens built from the code tokens in the interface and the body of the method. To vectorize each sub-token in  $Seq_c$ , we use a word embedding model, called GloVe [36], and treat each method as a sentence. After this vectorization, for the method, we obtain the sequence  $\langle v_1, v_2, \dots, v_n \rangle$  of the vectors of the sub-tokens in  $Seq_c$ . We then apply a sequential model on the sequence  $\langle v_1, v_2, \dots, v_n \rangle$  to learn the “summarized” vector  $V_{MC}$  that represents the method’s content. Specifically, we use Gated Recurrent Unit (GRU) [13], a type of RNN layer that is efficient in learning and capturing the information in a sequence.

2) *The method’s structure*: we first treat the method as the sequence of tokens and use GloVe to build the embeddings for all the tokens as in 1). We then replace every node in the AST of the method with the GloVe’s vector of the corresponding token of the node (Figure 6). From the tree of vectors, we use a tree-based model, called TreeCaps [12], to capture its structure to produce the “summarized” vector  $V_{AST}$  representing the method’s structure.

3) *Most similar faulty method*: for a method, we process the most similar buggy method  $M_b$  in the same way as the method’s structure via GloVe and TreeCaps to learn the vector  $V_{MSBM}$  for  $M_b$ .

Finally, for each method  $M_1$ , we obtain  $V_{MC}$ ,  $V_{AST}$ , and  $V_{MSBM}$ .

## 5.2 Statement-Level Representation Learning

Figure 7 shows how we build the vectors for a statement’s attributes.

1) *Code Coverage*: we directly use the vector  $V_{COV} = \langle c_1, c_2, \dots, c_K, r_1, r_2, \dots, r_K \rangle$  computed in Section 4.2 for the next computation.

2) *The statement’s structure*: we process the AST subtree representing the statement’s structure in the same manner (via GloVe and TreeCaps) as for the method’s structure to produce  $V_{subtree}$ .

3) *List of variables*: as with the method’s content, we run GloVe on the sequence of sub-tokens to produce a sequence of vectors and use GRU to produce the summarized vector  $V_{var}$  for the list.

Finally, for each statement  $S$ , we obtain  $V_{COV}$ ,  $V_{subtree}$ , and  $V_{var}$ .

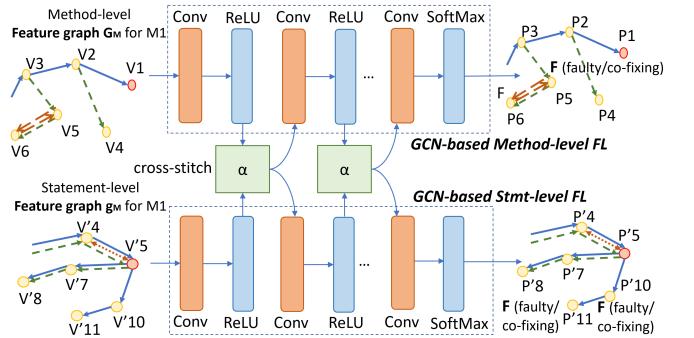


Figure 8: Dual-Task Learning Fault Localization

## 5.3 Feature Representation Learning

After computing the three embeddings for three attributes of each method, we use three fully connected layers to standardize each vector’s length to a chosen value  $l$ . Similarly, we use three fully connected layers for the three embeddings for each statement. Then, for a method or a statement, we concatenate the three output vectors from the fully connected layer to produce the vector  $V_M$  for the method and the other three vectors for  $V_S$  for the statement with the length of  $(l \times 3)$ .

After all, for a method  $M$ , we have the method-level graph  $G_M$  and the statement-level graph  $g_M$  with its statements. The nodes in  $G_M$  (Figure 4) now are the vectors computed for methods, and the nodes in  $g_M$  are the vectors  $V_S$  for the statements in  $M$  (Figure 5).

## 6 DUAL-TASK LEARNING FOR FAULT LOCALIZATION

Figures 8 and 9 illustrate our dual-task learning for fault localization. In the training dataset, for each bug  $B$ , to ensure the matching of a method and its corresponding statements, we build for each faulty method  $M$  the pairs  $(G_M, g_M)$ : 1)  $G_M$ , the method-level graph (Figure 4 with nodes replaced by vectors); and 2)  $g_M$ , the statement-level graph (Figure 5) containing all the statements belonging to  $M$ . To ensure the co-fixing connections among the buggy methods for the same bug  $B$ , we model the co-fixed methods of  $M$  via co-fixed relations in  $G_M$  (Figure 4). At the output layer, we label those methods as *faulty/co-fixing*. The co-fixed statements within  $g_M$  for the bug  $B$  are also labeled as *faulty/co-fixing*. The non-buggy methods or statements are labeled as *non-faulty*. The pairs  $(G_M, g_M)$  are used as the input of this dual-task learning model (Figure 8). We process all the faulty methods  $M$  for each bug  $B$ , and non-buggy methods.

In prediction, for each method  $M^*$  in the project, we build the pair  $(G_{M^*}, g_{M^*})$  and feed it to the trained dual-task model. In the output graphs, each node (for a method or a statement) will be classified as either *faulty/co-fixing* or *non-faulty*. The nodes with *faulty/co-fixing* labels in  $g_{M^*}$  are the co-fixing statements for the bug. Let us explain our dual-task learning in details.

### 6.1 Graph Convolutional Network (GCN) for FL

First, FixLOCATOR has two GCN models [19], each for FL at the method and statement levels. GCN processes the attributes of the nodes (vectors) and their edges (relations) in feature graphs. Each GCN model has  $n - 1$  pairs of a graph convolution layer ( $Conv$ ) and

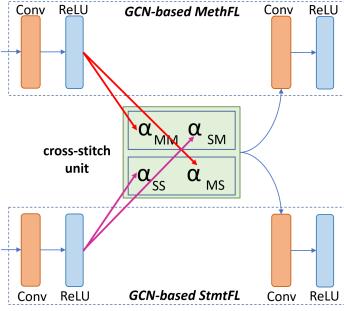


Figure 9: Dual-Task Learning via Cross-stitch Unit

a rectified linear unit (ReLU). They are aimed to consume and learn the characteristic features in the input feature graphs. The last pair of each GCN model is a pair of a graph convolution layer (Conv) and a softmax layer (softMax). The softMax layer plays the role of the classifier to determine whether a node for a method or a statement is labeled as *faulty/co-fixing* or *non-faulty*.

## 6.2 Dual-Task Learning with Cross-stitch Units

In a regular GCN model, those above pairs of Conv and ReLU are connected to one another. However, to achieve dual-task learning between method-level and statement-level FL (methFL and stmtFL), we apply a cross-stitch unit [30] to connect the two GCN models. The sharing of representations between methFL and stmtFL is modeled by learning a linear combination of the input features in both feature graphs  $G_M$  and  $g_M$ . At each of the ReLU layer of each GCN model (Figure 9), we aim to learn such a linear combination of the output from the graph convolution layers (Conv) of methFL and stmtFL.

The top sub-network in Figure 8 gets direct supervision from methFL and indirect supervision (through cross-stitch units) from stmtFL. Cross-stitch units regularize methFL and stmtFL by learning and enforcing shared representations by combining feature maps [30].

**Formulation.** For each pair of the GCN model, the outputs of the ReLU layer, called the hidden states, are computed as follows:

$$\hat{A} = D'^{-\frac{1}{2}} A' D' - \frac{1}{2} \quad (1)$$

$$H^i = \Delta(\hat{A} X^i W^i) \quad (2)$$

Where  $A'$  is the adjacency matrix of each feature graph;  $D'$  is the degree matrix;  $W^i$  is the weight matrix for layer  $i$ ;  $X^i$  is the input for layer  $i$ ;  $H^i$  is the hidden state of layer  $i$  and the output from the ReLU layer; and  $\Delta$  is the activation function ReLU. In a regular GCN,  $H^i$  is the input of the next layer of GCN (i.e., the input of Conv).

In Figures 8 and 9, a cross-stitch unit is inserted between the ReLU layer of the previous pair and the Conv layer of the next one. The input of the cross-stitch unit includes the outputs of the two ReLU layers:  $H_M^i$  and  $H_S^i$  (i.e., the hidden states of those layers in methFL and stmtFL). We aim to learn the linear combination of both inputs of the cross-stitch unit, which is parameterized using the weights  $\alpha$ . Thus, the output of the cross-stitch unit is computed as:

$$\begin{bmatrix} X_M^{i+1} \\ X_S^{i+1} \end{bmatrix} = \begin{bmatrix} \alpha_{MM} & \alpha_{MS} \\ \alpha_{SM} & \alpha_{SS} \end{bmatrix} \begin{bmatrix} H_M^i \\ H_S^i \end{bmatrix} \quad (3)$$

$\alpha$  is the trainable weight matrix;  $X_M^{i+1}$  and  $X_S^{i+1}$  are the inputs for the  $(i+1)^{th}$  layers of the GCNs at the method and statement levels.

$X_M^{i+1}$  and  $X_S^{i+1}$  contain the information learned from both methFL and stmtFL, which helps achieve the main goal for dual-task learning to enhance the performance of fault localization at both levels.

In general,  $\alpha$ s can be set. If  $\alpha_{MS}$  and  $\alpha_{SM}$  are set to zeros, the layers are made to be task-specific. The  $\alpha$  values model linear combinations of feature maps. Their initialization in the range [0,1] is important for stable learning, as it ensures that values in the output activation map (after cross-stitch unit) are of the same order of magnitude as the input values before linear combination [30].

If the sizes of the  $H_M^i$  and  $H_S^i$  are different, we need to adjust the sizes of the matrices. From Formula 3, we have:

$$X_M^{i+1} = \alpha_{MM} H_M^i + \alpha_{MS} H_S^i \quad (4)$$

$$X_S^{i+1} = \alpha_{SM} H_M^i + \alpha_{SS} H_S^i \quad (5)$$

We resize  $H_S^i$  in Formula 4 and resize  $H_M^i$  in Formula 5 if needed. We use the *bilinear interpolation* technique [39] in image processing for resizing. We pad zeros to the matrix to make the aspect ratio 1:1. If the size needs to be reduced, we do the center crop on the matrix to match the required size.

FixLOCATOR also has a trainable threshold for softMax to classify if a node corresponding to a method or a statement is faulty or not.

## 7 EMPIRICAL EVALUATION

### 7.1 Research Questions

For evaluation, we seek to answer the following research questions:

**RQ1. Comparison with State-of-the Art Deep Learning (DL)-based Approaches.** How well does FixLOCATOR perform compared with the state-of-the-art DL-based fault localization approaches?

**RQ2. Impact Analysis of Dual-Task Learning.** How does the dual-task learning scheme affect FixLOCATOR's performance?

**RQ3. Sensitivity Analysis.** How do various factors affect the overall performance of FixLOCATOR?

**RQ4. Evaluation on Python Projects.** How does FixLOCATOR perform on Python code?

**RQ5. Extrinsic Evaluation on Usefulness.** How much does FixLOCATOR help an APR tool improve its bug-fixing?

### 7.2 Experimental Methodology

**7.2.1 Dataset.** We use a benchmark dataset Defects4J V2.0.0 [1] with 835 bugs from 17 Java projects. For each bug in a project  $P$ , Defects4J has the faulty and fixed versions of the project. The faulty and fixed versions contain the corresponding test suite relevant to the bug. With the Diff comparison between faulty and fixed versions of a project, we can identify the faulty statements. Specifically, for a bug in  $P$ , Defects4J has a separate copy of  $P$  but with only the corresponding test suite revealing the bug. For example,  $P_1$ , a version of  $P$ , passes a test suite  $T_1$ . Later, a bug  $B_1$  in  $P_1$  is identified. After debugging,  $P_1$  has an evolved test suite  $T_2$  detecting the bug. In this case, Defects4J has a separate copy of the buggy  $P_1$  with a single bug, together with the test suite  $T_2$ . Similarly, for bug  $B_2$ , Defects4J has a copy of  $P_2$  together with  $T_3$  (evolving from  $T_2$ ), and so on. We do not use the whole  $T$  of all test suites for training/testing. For within-project setting, we test one bug  $B_i$  with test suite  $T_{(i+1)}$  by

training on all other bugs in  $P$ . We conducted all the experiments on a server with 16 core CPU and a single Nvidia A100 GPU.

In Defects4J-v2.0, regarding the statistics on the number of buggy/fixed statements for a bug, there are 199 bugs with one buggy/fixed statement, 142 bugs with two, 90 bugs with three, 78 bugs with four, 43 bugs with five, and 283 bugs with  $>5$  buggy statements. Regarding the statistics on the number of buggy/fixed methods/hunks for a bug, there are 199 bugs with one-method/one-statement, 105 bugs with one-method/multi-statements, 142 bugs with multi-methods/one-statement for each method, 61 bugs with multi-methods/multi-statements for each method, and 357 bugs with multiple methods, each has one or multiple buggy statements. Thus, there are 665 (out of 864 bugs) with CC fixing statements.

### 7.2.2 Experimental Setup and Procedures.

#### RQ1. Comparison with DL-based FL Approaches.

*Baselines.* Our tool aims to output a set of CC fixing statements for a bug. However, the existing Deep Learning-based FL approaches can produce only the ranked lists of suspicious statements with scores. Thus, we chose as baselines the most recent, state-of-the-art, DL-based, statement-level FL approaches: (1) CNN-FL [50]; (2) DeepFL [22]; and (3) DeepRL4FL [24]; then, we use the predicted, ranked list of statements as the output set. For the comparison in ranking with those ranking baselines, we convert our tool’s result into a ranked list by ranking the statements in the predicted set by the classification scores (i.e., before deriving the final set). We also compare with the CC fixing-statement detection module in DEAR [25], a multi-method/multi-statement APR tool.

*Procedures.* We use the leave-one-out setting as in prior work [22, 23] (i.e., testing on one bug and training on all other bugs). We also consider the order of the bugs in the same project via the revision numbers. Specifically, for each buggy version  $B$  of project  $P$  in Defects4J, all buggy versions from the other projects are first included in the training data. Besides, we separate all the buggy versions of the project  $P$  into two groups: 1) one buggy version as the test data for model prediction, and 2) all the buggy versions of the same project  $P$  that have occurred before the buggy version  $B$  are also included in the training data. If the latter group is empty, only the buggy versions from the other projects are used for training to predict for the current buggy version in  $P$ .

We tune all models using autoML [5] to find the best parameter setting. We directly follow the baseline studies to select the parameters that need to be tuned in the baselines. We tuned our model with the following key hyper-parameters to obtain the best performance: (1) Epoch size (i.e., 100, 200, 300); (2) Batch size (i.e., 64, 128, 256); (3) Learning rate (i.e., 0.001, 0.003, 0.005, 0.010); (4) Vector length of word representation and its output (i.e., 150, 200, 250, 300); (5) The output channels of convolutional layer (16, 32, 64, 128); (6) The number of convolutional layers (3, 5, 7, 9).

DeepFL was proposed for the method-level FL. For comparison, following a prior study [24], we use only DeepFL’s spectrum-based and mutation-based features applicable to detect faulty statements.

*Evaluation Metrics.* We use the following metrics for evaluation:

(1) **Hit-N** measures the number of bugs that the predicted set contains at least  $N$  faulty statements (i.e., the predicted and oracle sets for a bug overlap at least  $N$  statements regardless of the sizes of both sets). Both precision and recall can be computed from Hit-N.

(2) **Hit-All** is the number of bugs in which the predicted set covers the correct set in the oracle for a bug.

(3) **Hit-N@Top-K** is the number of bugs that the predicted list of the top- $K$  statements contains at least  $N$  faulty statements. This metric is used when we compare the approaches in ranking.

#### RQ2. Impact Analysis of Dual-Task Learning Model.

*Baselines.* To study the impact of dual-task learning, we built two variants of FixLOCATOR: (1) *Statement-only model*: the method-level FL model (`methFL`) is removed from FixLOCATOR and only statement-level FL (`stmtFL`) is kept for training. (2) *Cascading model*: in this variant, dual-task learning is removed, and we cascade the output of `methFL` directly to the input of `stmtFL`.

*Procedures.* The statement-only model has only the statement-level fault localization. We ran it on all methods in the project to find the faulty statements. We use the same training strategy and parameter tuning as in RQ1. We use Hit-N for evaluation.

**RQ3. Sensitivity Analysis.** We conduct ablation analysis to evaluate the impact of different factors on the performance: every node feature, co-change relation, and the depth limit on the stack trace and the execution trace. Specifically, we set FixLOCATOR as the complete model, and each time we built a variant by removing one key factor, and compared the results. Except for the removed factor, we keep the same setting as in other experiments.

**RQ4. Evaluation on Python Projects.** To evaluate FixLOCATOR on different programming languages, we ran it on the Python benchmark BugsInPy [2, 40] with 441 bugs from 17 different projects.

**RQ5. Extrinsic Evaluation.** To evaluate usefulness, we replaced the original CC fixing-location module in DEAR [25] with FixLOCATOR to build a variant of DEAR, namely DEAR<sup>FixL</sup>. We also added FixLOCATOR and Ochiai FL [7] to CURE [15] to build two variants: CURE<sup>FixL</sup> (FixLOCATOR + CURE) and CURE<sup>Ochi</sup> (Ochiai+CURE).

## 8 EMPIRICAL RESULTS

### 8.1 RQ1. Comparison Results with State-of-the-Art DL-based FL Approaches

Table 1 shows how well FixLOCATOR’s coverage is on the actual correct CC fixing statements (recall). The result is w.r.t. the bugs in the oracle with different numbers  $K$  of CC fixing statements:  $K = \#CC\text{-}Stmts = 1, 2, 3, 4, 5$ , and  $5+$ . For example, in the oracle, there are 90 bugs with 3 faulty statements. FixLOCATOR’s predicted set correctly contains all 3 buggy statements for 21 bugs (Hit-All), 2 of them for 25 bugs, and 1 faulty statement for 51 bugs. As seen, regardless of  $N$ , FixLOCATOR performs better in any Hit- $N$  over the baselines for all  $K$ s. Note that Hit-All = Hit- $N$  when  $N(\#overlaps) = K(\#CC\text{-}Stmts)$ .

Table 2 shows the summary of the comparison results in which we sum all the corresponding Hit- $N$  values across different numbers  $K$  of CC fixing statements in Table 1. As seen, FixLOCATOR can improve CNN-FL, DeepFL, DeepRL4FL, and DEAR by 16.6%, 16.9%, 9.9%, and 20.6%, respectively, in terms of Hit-1 (i.e., the predicted set contains at least one faulty statement). It also improves over those baselines by 33.6%, 40.3%, 26.5%, and 57.5% in terms of Hit-2, 43.9%, 46.4%, 28.1%, and 51.9% in terms of Hit-3, 100%, 155.6%, 64.5%, and 142.1% in terms of Hit-4. Note: Any Hit- $N$  reflects the cases of multiple CC statements. For example, Hit-1 might include the bugs with more than one buggy/fixed statement. Importantly, our tool

**Table 1: RQ1. Detailed Comparison w.r.t. Faults with Different # of CC Fixing Statements in an Oracle Set (Recall)**

#CC-Stmts in Oracle	Metrics	CNN-FL	DeepFL	DeepRL4FL	DEAR	Fix-LOCATOR
1 (199 bugs)	Hit-1	78	76	84	74	93
2 (142 bugs)	Hit-1	67	64	70	65	75
	Hit-2	33	30	34	28	41
3 (90 bugs)	Hit-1	46	44	47	42	51
	Hit-2	21	20	23	20	25
	Hit-3	11	10	13	12	21
4 (78 bugs)	Hit-1	41	42	42	40	45
	Hit-2	22	19	21	20	24
	Hit-3	9	7	8	5	12
	Hit-4	3	2	4	2	9
5 (43 bugs)	Hit-1	15	14	16	13	18
	Hit-2	9	8	9	7	12
	Hit-3	6	5	6	5	7
	Hit-4	3	2	3	2	3
	Hit-5	1	1	1	0	1
5+ (283 bugs)	Hit-1	85	91	93	87	105
	Hit-2	40	42	45	41	65
	Hit-3	31	34	37	32	42
	Hit-4	17	14	21	15	34
	Hit-5	4	3	5	2	8
	Hit-5+	1	2	3	1	3

**Table 2: RQ1. Comparison Results with DL-based FL Models**

Metrics	CNN-FL	DeepFL	DeepRL4FL	DEAR	FixLOCATOR
Hit-1	332	331	352	321	387
Hit-2	125	119	132	106	167
Hit-3	57	56	64	54	82
Hit-4	23	18	28	19	46
Hit-5	5	4	6	2	9
Hit-5+	1	2	3	1	3
Hit-All	127	121	139	115	168

produced the exact-match sets for 168/864 bugs (19.5%), relatively improving over the baselines 32%, 38.8%, 20.8%, and 46.1% in Hit-All. It performs well in Hit-All when the number of CC statements  $K=1\text{--}4$ . However, producing the exact-matched sets for all statements when  $K \geq 5$  is still challenging for all the models.

Table 3 shows the comparison on *how precise the results are* in a predicted set. For example, when *the number of the CC statements in a predicted set is  $K'=3$* , there are 23 bugs in which all of those 3 faulty statements are correct (there might be other statements missing). There are 27 bugs in which two of the 3 predicted, faulty statements are correct. There are 55 bugs in which only one of the 3 predicted, faulty statements are correct. As seen, regardless of  $N$ , FixLOCATOR is more precise than the baselines for all  $K'$ .

Table 4 shows the comparison as ranking is considered (Hit-N@Top-K). As seen, in the ranking setting, FixLOCATOR locates more CC fixing statements than any baseline. For example, FixLOCATOR improves the best baseline DeepRL4RL by 23.9% in Hit-2@Top-5, 22.6% in Hit-3@Top-5, 43.8% in Hit-4@Top-5, and 22.2% in Hit-5@Top-5, respectively. The same trend is for Hit-N@Top-10.

We did not compare with the spectrum-/mutation-based FL models since DeepRL4FL [24] was shown to outperform them.

**Table 3: RQ1. Detailed Comparison w.r.t. Faults with Different # of CC Fixing Statements in a Predicted Set (Precision)**

#Stmts in Predicted Set	Metrics	CNN-FL	DeepFL	DeepRL4FL	DEAR	Fix-LOCATOR
1 (203 bugs)	Hit-1	83	79	87	75 (183)	99
2 (165 bugs)	Hit-1	75	72	78	71 (172)	83
	Hit-2	36	34	39	34 (172)	45
3 (120 bugs)	Hit-1	52	46	48	41 (129)	55
	Hit-2	24	22	26	19 (129)	27
	Hit-3	12	11	14	10 (129)	23
4 (96 bugs)	Hit-1	47	49	46	33 (78)	51
	Hit-2	24	21	22	14 (78)	26
	Hit-3	11	9	10	5 (78)	14
	Hit-4	5	3	6	1 (78)	11
5 (73 bugs)	Hit-1	17	16	17	12 (55)	19
	Hit-2	10	10	11	7 (55)	14
	Hit-3	8	6	7	4 (55)	9
	Hit-4	3	3	4	1 (55)	5
	Hit-5	2	1	2	0 (55)	2
5+ (178 bugs)	Hit-1	58	69	76	68(218)	80
	Hit-2	31	32	34	32(218)	55
	Hit-3	26	30	33	24(218)	36
	Hit-4	15	12	18	16(218)	30
	Hit-5	3	3	4	5 (218)	7
	Hit-5+	1	2	3	2 (218)	3

**Table 4: RQ1. Comparison with Baselines w.r.t. Ranking**

N=	Hit-N@Top-5					Hit-N@Top-10					
	1	2	3	4	5	1	2	3	4	5	5+
CNN-FL	533	311	133	33	4	578	386	166	42	10	81
DeepFL	525	298	131	35	6	563	364	156	42	10	83
DeepRL4FL	586	339	159	32	9	623	407	186	48	13	92
DEAR	501	274	119	25	3	544	341	142	36	7	71
FixLOCATOR	633	420	195	46	11	690	470	217	51	13	94

**Table 5: Overlapping Analysis Results for Hit-1**

	FixLOCATOR		
	Unique-Baseline	Overlap	Unique-FixLOCATOR
CNN-FL	48	284	103
DeepFL	54	277	110
DeepRL4FL	61	291	96
DEAR	35	286	101

We also performed the analysis on the overlapping between the results of FixLOCATOR and each baseline. As seen in Table 5, FixLOCATOR can detect at least one correct faulty statement in 103 bugs that CNN-FL missed, while CNN-FL can do so only in 48 bugs that FixLOCATOR missed. Both FixLOCATOR and CNN-FL can do so in the same 284 bugs. In brief, FixLOCATOR can detect at least one correct buggy statement in more “unique” bugs than any baseline.

## 8.2 RQ2. Impact Analysis Results on Dual-Task Learning

Table 6 shows that FixLOCATOR has better performance in detecting CC fixing statements than the two variants (statement-only and cascading models). This result shows that the **dual-task learning helps improve FL over the cascading model** ( $\text{methFL} \rightarrow \text{stmtFL}$ ).

**Table 6: RQ2. Impact Analysis of Dual-Task Learning**

Variant	Hit-1	Hit-2	Hit-3	Hit-4	Hit-5	Hit-5+
Stmt-only	304	111	51	11	3	2
Cascading	343	125	61	19	3	3
FixLOCATOR	387	167	82	46	9	3

**Table 7: RQ3. Sensitivity Analysis of Method- and Statement-Level Features. ML: Method-level; SL: Statement-level**

Model Variant		Hit-N					
		1	2	3	4	5	5+
ML	w/o Method Content	366	158	78	39	9	3
	w/o Method Structure	357	155	80	40	8	3
	w/o Similar Buggy Method	361	157	79	44	9	3
	w/o ML Co-change Rel.	355	152	77	40	8	3
SL	w/o Code Coverage	348	151	75	38	7	2
	w/o AST Subtree	354	153	77	41	8	3
	w/o Variables	373	162	78	42	9	3
	w/o SL Co-change Relation	351	150	76	39	7	2
	FixLOCATOR	387	167	82	46	9	3

**Table 8: RQ3. Sensitivity Analysis (Depth of Traces)**

Depth	Hit-N					
	1	2	3	4	5	5+
5	371	162	74	42	8	3
<b>10</b>	387	167	82	46	9	3
15	368	158	71	39	7	3

Moreover, without the impact of method-level FL (`methFL`), the performance decreases significantly, indicating `methFL`'s contribution.

### 8.3 RQ3. Sensitivity Analysis Results

**8.3.1 Impact of the Method-Level (ML) Features and ML Co-change Relation.** Among all the method-level features/attributes of FixLOCATOR, the *feature of co-change relations among methods has the largest impact*. Specifically, without the co-change feature among methods, Hit-1 is decreased by 8.3%. Moreover, the *method structure feature, represented as AST, has the second largest impact*. Without the method structure feature, Hit-1 is decreased by 7.8%.

Among the last two method-level features with least impact, the method content feature has less impact than the similar-buggy-method feature. This shows that *the bugginess nature of a method and similar ones has more impact than the tokens of the method itself*.

**8.3.2 Impact of the Statement-Level (SL) Features and SL Co-change Relation.** Among all the statement-level features, *Code Coverage has the largest impact*. Without *Code Coverage* feature, Hit-1 is decreased by 10.1%. The co-change relations among statements have the second largest impact among all SL features/attributes. Specifically, without the co-change relations among statements, Hit-1 is decreased by 9.3%.

**8.3.3 Impact of the Depth Level of Stack Trace.** As seen in Table 8, FixLOCATOR can achieve the best performance when depth=10. The cases with depth= 5 or 15 can bring into analysis too few or too many irrelevant methods, causing more noises to the model. Thus, we chose depth=10 for our experiments.

```

1 public UnivariateRealValuePair optimize(final FUNC f, GoalType goal,
2     double min, double max) throws FunctionEvaluationException {
3     - return optimize(f, goal, min, max, 0);
4     + return optimize(f, goal, min, max, min + 0.5 * (max - min));
5 }
6     ...
7     try {
8     - final double bound1 = (i == 0) ? min : min + generator.nextDouble()...
9     - final double bound2 = (i == 0) ? max : min + generator.nextDouble()...
10    - optima[i] = optimizer.optimize(f, goal, FastMath.min(bound1, bound2)...;
11    + final double s = (i == 0) ? minValue : min + generator.nextDouble()...
12    + optima[i] = optimizer.optimize(f, goal, min, max, s); ...
13 }
```

**Figure 10: An Illustrating Example****Table 9: Ranking of CC Fixing Locations for Figure 10**

LOC	CNN-FL	DeepFL	DeepRL4FL	DEAR	FixLOCATOR
Line 2	<b>1</b>	22	2	27	★ (no rank)
Line 8	24	<b>3</b>	6	12	★ (no rank)
Line 9	25	<b>4</b>	7	13	★ (no rank)
Line 10	50+	13	16	39	★ (no rank)

**Table 10: RQ4. BugsInPy (Python Projects) versus Defects4J (Java Projects). P% = |Located Bugs|/|Total Bugs in Datasets|**

Metrics	BugsInPy (Python projects)		Defects4J (Java projects)	
	P%	Cases	P%	Cases
Hit-1	43.8%	193	46.3%	387
Hit-2	16.3%	72	20.0%	167
Hit-3	10.2%	45	9.8%	82
Hit-4	3.4%	15	5.5%	46
Hit-5	0.7%	3	1.1%	9
Hit-5+	0%	0	0.4%	3

**8.3.4 Illustrating Example.** Table 9 displays the ranking from the models for Figure 10. FixLOCATOR correctly produces all 4 CC fixing statements in its predicted set (lines 2,8,9, and 10 in two methods). The statement-only model detects only line 2 as faulty. It completely missed lines 8–10 of the `optimize` method. In contrast, the cascading model detects lines 8–10, however, its `MethFL` considers the first method (`optimize(...)` at line 1) as non-faulty, thus, it did not detect the buggy line 2 due to its cascading.

The baselines CNN-FL, DeepFL, DeepRL4FL, and DEAR detect only 1, 2, 1, and 0 faulty statements (bold cells) in their top-4 resulting lists, respectively. In brief, the baselines are not designed to detect CC fixing locations, thus, their top-K lists are not correct.

### 8.4 RQ4. Evaluation on Python Projects

As seen in Table 10, FixLOCATOR can localize 193 faulty statements with Hit-1. This shows that the performance on the Python projects is consistent with that on the Java projects. Specifically, at the statement level, the percentages of the total Python and Java bugs that can be localized are similar, e.g., 43.8% vs. 46.3% with Hit-1.

### 8.5 RQ5. Extrinsic Evaluation: Usefulness in Automated Program Repair

In Table 11, with its better CC fixing-locations, FixLOCATOR can help DEAR<sup>FixL</sup> relatively improve over DEAR in auto-fixing 10.5%

**Table 11: RQ5. Usefulness in APR (running on Defects4J)**

Bug Types	1-Meth 1-Stmt	1-Meth M-Stmt	M-Meths 1-Stmt	M-Meths M-Stmts	M-Meths Mix-Stmts	Total
DEAR	64	12	24	2	3	105
DEAR <sup>FixL</sup>	69	13	26	2	4	116
CURE <sup>Ochi</sup>	84	0	0	0	0	84
CURE <sup>FixL</sup>	79	11	26	1	3	120

**Table 12: RQ5. Detailed Results on Usefulness in APR**

Projects in Defects4J	DEAR	DEAR <sup>FixL</sup>	CURE <sup>Ochi</sup>	CURE <sup>FixL</sup>
Chart	8/16	9/18	6/13	9/19
Cli	5/11	7/14	4/11	7/16
Closure	7/11	8/13	6/10	9/14
Codec	1/4	1/4	1/4	1/4
Collections	0/0	0/0	0/0	0/0
Compress	7/15	7/16	5/12	8/17
Csv	2/5	2/6	1/4	2/5
Gson	1/3	1/3	1/2	1/3
JacksonCore	4/9	5/10	3/7	6/9
JacksonDatabind	16/27	17/29	13/25	16/28
JacksonXml	1/1	1/1	1/1	1/1
Jsoup	13/21	14/22	10/17	16/23
JxPath	8/14	9/15	6/13	10/17
Lang	8/15	9/17	9/16	11/15
Math	20/33	20/31	16/25	20/35
Mockito	1/2	1/2	1/1	1/2
Time	3/6	3/6	1/3	3/6
Total	105/193	116/207	84/164	120/214

X/Y: are the numbers of correct and plausible patches; Dataset: Defects4J

**Table 13: Running Time**

Models	CNN-FL	DeepFL	DeepRL4FL	DEAR	FixLOCATOR
Training Time	4 hours	5 mins	7 hours	21 hours	6 hours
Prediction Time	2 seconds	1 second	4 seconds	9 seconds	2 seconds

more bugs (11 bugs) across all bug types. Moreover, CURE<sup>FixL</sup> (FixLOCATOR+ CURE) can fix 42.9% relatively more bugs (36 bugs) than CURE<sup>Ochi</sup> (Ochiai+CURE). Especially, CURE<sup>FixL</sup> fixed 41 *more bugs with multi-statements or multi-methods*. The 5 bugs with single buggy statements that CURE<sup>FixL</sup> missed are due to FixLOCATOR incorrectly producing more than one fixing locations. Table 12 shows that FixLOCATOR can help DEAR and CURE improve both correct and plausible patches (passing all the tests) across all projects.

## 8.6 Further Analysis

**8.6.1 Running Time.** As seen in Table 13, except for DeepFL (using a basic neural network), the other approaches have similar training and prediction time. Importantly, prediction time is just a few seconds, making FixLOCATOR suitable for interactive use.

**8.6.2 Limitations.** First, our tool does not detect well the sets with +5 CC fixing statements since it does not learn well those large co-changes. Second, it does not work in locating a fault that require only adding statements to fix (neither do all baselines). Third, if the faulty statements/methods occur far from the crash method in the execution traces, it is not effective. Finally, it does not have any mechanism to integrate program analysis in expanding the faulty statements having dependencies with the detected faulty ones.

**8.6.3 Threats to Validity.** (1) We evaluated FixLOCATOR on Java and Python. Our modules are general for any languages. (2) We compared the models only on two datasets that have test cases. (3) For

comparison, we use only DeepFL’s features applicable to statement-level FL although it works at the method level. Other baselines work directly at the statement level. (4) In 501 bugs in BugsInPy, the third-party tool cannot process 60 of them. (5) We focus on CC fixing statements, instead of methods, due to bug fixing purpose.

## 9 RELATED WORK

Several types of techniques have been proposed to locate faulty statements/methods. However, none of the existing FL approaches detect *CC fixing locations*. A related work is DEAR [25], which uses a combination of BERT and data flows to locate CC statements. Hercules APR tool [37] can detect multiple buggy hunks of code. It can detect only the buggy hunks with similar statements (replicated fixes), while our tool detects general CC fixing locations. In comparison, FixLOCATOR and Hercules detect 26 and 15 multi-hunk bugs respectively among 395 bugs in Defects4J-v1.2 [37].

The Spectrum-based Fault Localization (SBFL) [6, 8, 16, 27, 29, 33, 43, 46] and Mutation-based Fault Localization (MBFL) [11, 31, 32, 35, 47, 48] have been proposed for statement-level FL. Their key limitations are that they cannot differentiate the statements with the same scores or cannot have effective mutators to catch a complex fault. Among the learning-based FL models, learning-to-rank FL approaches [9, 23, 38, 45] aim to locate faulty methods. Statistical FL has been combined with causal inference for statement-level FL [20]. All of those models do not locate CC fixing statements.

Machine learning has also been used for FL. Early neural network-based FL [10, 42, 51, 53] mainly use test coverage data. A limitation is that they cannot distinguish elements accidentally executed by failed tests and the actual faulty elements [23]. Deep learning-based approaches, GRACE [28], DeepFL [22], CNNFL [50], DeepRL4FL [24] achieve better results. GRACE [28] proposes a new graph representation for a method and learns to rank the faulty methods. In contrast, FixLOCATOR is aimed to locate multiple CC fixing statements in a fix for a fault. DeepFL and DeepRL4FL can outperform the learning-based and early neural networks FL techniques, such as MULTRIC [45], TrapT [23], and Fluccs [38]. In our empirical evaluation, we showed that FixLOCATOR can outperform those baselines under study in detecting CC fixing statements.

## 10 CONCLUSION

We present FixLOCATOR, a novel DL-based FL approach that aims to locate co-change fixing locations within one or multiple methods. The key ideas of FixLOCATOR include (1) a new dual-task learning model of method- and statement-level fault localization to detect CC fixing locations; (2) a novel graph-based representation learning with co-change relations among methods and statements; (3) a novel feature in co-change methods/statements. Our empirical results show that FixLOCATOR relatively improves over the state-of-the-art FL baselines by locating more *CC fixing statements* from 26.5% to 155.6%, and help APR tools improve its bug-fixing accuracy.

## ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CNS-2120386, CCF-1723215, CCF-1723432, TWC-1723198, and the US National Security Agency (NSA) grant NCAEC-002-2021 on Cybersecurity Research Innovation.

## REFERENCES

- [1] 2019. *The Defects4J Data Set*. <https://github.com/rjust/defects4j>
- [2] 2020. *The BugsInPy Data Set*. <https://github.com/soarsmu/BugsInPy>
- [3] 2021. FixLocator. <https://github.com/fixlocator/research/fixlocatorresearch>
- [4] 2021. JDT. <https://www.eclipse.org/jdt/core/tools/jdtcoretools/index.php>
- [5] 2021. *The NNI autoML tool*. <https://github.com/microsoft/nni>
- [6] 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.
- [7] 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.
- [8] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.
- [9] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 177–188.
- [10] Lionel C Briand, Yvan Labiche, and Xuetao Liu. 2007. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 137–146.
- [11] Timothy Alan Budd. 1981. MUTATION ANALYSIS OF PROGRAM TEST DATA. (1981).
- [12] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 30–38.
- [13] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).
- [14] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülcöhre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR abs/1406.1078* (2014). [arXiv:1406.1078](https://arxiv.org/abs/1406.1078) <http://arxiv.org/abs/1406.1078>
- [15] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering*. 1161–1173. <https://doi.org/10.1109/ICSE43902.2021.00107>
- [16] 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 (ASE'05)*. ACM, 273–282.
- [17] J. A. Jones, M. J. Harrold, and J. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*. 467–477. <https://doi.org/10.1145/581396.581397>
- [18] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*. IEEE, 114–125.
- [19] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
- [20] Yiğit Küçük, Tim AD Henderson, and Andy Podgurski. 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 649–660.
- [21] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for \$8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 3–13.
- [22] Xia Li, Wei Li, Yiqun 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*. ACM, 169–180.
- [23] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.
- [24] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*. IEEE.
- [25] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. ACM Press.
- [26] Ziyao Li, Liang Zhang, and Guojie Song. 2019. GCN-LASE: Towards adequately incorporating link attributes in graph convolutional networks. *arXiv preprint arXiv:1902.09817* (2019).
- [27] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) (PLDI '05). Association for Computing Machinery, New York, NY, USA, 15–26. <https://doi.org/10.1145/1065010.1065014>
- [28] 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.
- [29] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdinand Thung, and Aditya Budi. 2014. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process* 26, 2 (2014), 172–219.
- [30] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. 2016. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3994–4003.
- [31] S. Moon, Y. Kim, M. Kim, and S. Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *IEEE International Conference on Software Testing, Verification and Validation*. 153–162. <https://doi.org/10.1109/ICST.2014.28>
- [32] Vincenzo Musco, Martin Monperrus, and Philippe Preux. 2017. A large-scale study of call graph-based impact prediction using mutation testing. *Software Quality Journal* 25, 3 (2017), 921–950.
- [33] 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), 11.
- [34] Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate "unknown" faults. In *IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 691–700.
- [35] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5–7 (2015), 605–628.
- [36] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. <http://www.aclweb.org/anthology/D14-1162>
- [37] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. IEEE Press. <https://doi.org/10.1109/ICSE.2019.00020>
- [38] 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.
- [39] Unknown. 2022. Bilinear Interpolation. [https://en.wikipedia.org/wiki/Bilinear\\_interpolation](https://en.wikipedia.org/wiki/Bilinear_interpolation)
- [40] Ratnadirna Widayarsi, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1556–1560.
- [41] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Softw. Eng.* 42, 8 (Aug. 2016), 707–740. <https://doi.org/10.1109/TSE.2016.2521368>
- [42] W Eric Wong and Yu Qi. 2009. BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering* 19, 04 (2009), 573–597.
- [43] W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. IEEE, 449–456.
- [44] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sungjun Kim. 2014. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 204–214.
- [45] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 191–200.
- [46] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*. IEEE, 23–32.
- [47] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *IEEE International Conference on Software Maintenance (ICSM'10)*. IEEE, 1–10.
- [48] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting Mechanical Faults to Localize Developer Faults for Evolving Software. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (Indianapolis, Indiana, USA) (OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 765–784. <https://doi.org/10.1145/2509136.2509551>
- [49] Zhenyu Zhang, Wing Kwong Chan, TH Tse, Bo Jiang, and Xinming Wang. 2009. Capturing propagation of infected program states. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 43–52.

- [50] 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.
- [51] Zhuo Zhang, Yan Lei, Qingping Tan, Xiaoguang Mao, Ping Zeng, and Xi Chang. 2017. Deep Learning-Based Fault Localization with Contextual Information. *Ieice Transactions on Information and Systems* 100, 12 (2017), 3027–3031.
- [52] Lei Zhao, Lina Wang, Zuoting Xiong, and Dongming Gao. 2010. Execution-aware fault localization based on the control flow analysis. In *International Conference on Information Computing and Applications*. Springer, 158–165.
- [53] Wei Zheng, Desheng Hu, and Jing Wang. 2016. Fault localization analysis based on deep neural network. *Mathematical Problems in Engineering* 2016 (2016). <https://doi.org/10.1155/2016/1820454>