PAFL: Enhancing Fault Localizers by Leveraging Project-Specific Fault Patterns

DONGUK KIM, Korea University, Republic of Korea
MINSEOK JEON, Korea University, Republic of Korea
DOHA HWANG, Samsung Electronics, Republic of Korea
HAKJOO OH, Korea University, Republic of Korea

We present PAFL, a new technique for enhancing existing fault localization methods by leveraging project-specific fault patterns. We observed that each software project has its own challenges and suffers from recurring fault patterns associated with those challenges. However, existing fault localization techniques use a universal localization strategy without considering those repetitive faults. To address this limitation, our technique, called project-aware fault localization (PAFL), enables existing fault localizers to leverage project-specific fault patterns. Given a buggy version of a project and a baseline fault localizer, PAFL first mines the fault patterns from past buggy versions of the project. Then, it uses the mined fault patterns to update the suspiciousness scores of statements computed by the baseline fault localizer. To this end, we use two novel ideas. First, we design a domain-specific fault pattern-description language to represent various fault patterns. An instance, called crossword, in our language describes a project-specific fault pattern and how it affects the suspiciousness scores of statements. Second, we develop an algorithm that synthesizes crosswords (i.e., fault patterns) from past buggy versions of the project. Evaluation using seven baseline fault localizers and 12 real-world C/C++ and Python projects demonstrates that PAFL effectively, robustly, and efficiently improves the performance of the baseline fault localization techniques.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Domain specific languages.*

Additional Key Words and Phrases: Fault Localization, Domain-Specific Language, Program Synthesis

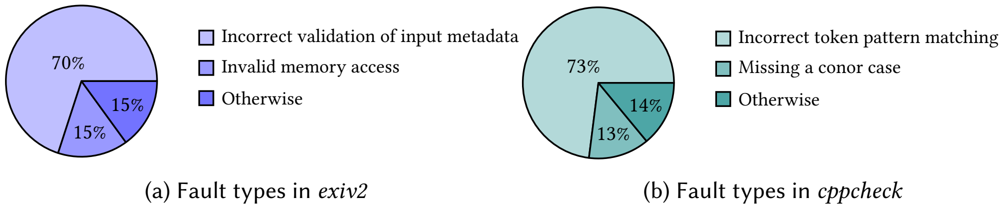
ACM Reference Format:
Donguk Kim, Minseok Jeon, Doha Hwang, and Hakjoo Oh. 2025. PAFL: Enhancing Fault Localizers by Leveraging Project-Specific Fault Patterns. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 129 (April 2025), 28 pages. https://doi.org/10.1145/3720526

1 Introduction

Fault localization is a crucial software engineering technique that aims to automatically identify the root causes of faults in programs. Over the past decades, various fault localization techniques have been proposed, including spectrum-based (SBFL) [Abreu et al. 2006; Jones et al. 2002], mutation-based (MBFL) [Moon et al. 2014; Papadakis and Le Traon 2015], deep learning-based (DLFL) [Li et al. 2019; Lou et al. 2021], and LLM-based [Yang et al. 2024] fault localization techniques. These methods can significantly reduce the burden on developers and serve as a key ingredient in other software engineering techniques, such as automatic program repair [Gazzola et al. 2019; Liu et al.

Authors' Contact Information: Donguk Kim, Korea University, Seoul, Republic of Korea, donguk_kim@korea.ac.kr; Minseok Jeon, Korea University, Seoul, Republic of Korea, minseok_jeon@korea.ac.kr; Doha Hwang, Samsung Electronics, Suwon, Republic of Korea, doha.hwang@samsung.com; Hakjoo Oh, Korea University, Seoul, Republic of Korea, hakjoo_oh@korea.ac.kr.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/4-ART129
https://doi.org/10.1145/3720526



(a) Fault types in *exiv2*

(b) Fault types in *cppcheck*

Fig. 1. Distribution of faults in *exiv2* and *cppcheck*.

2019; Xia and Zhang 2022; Ye and Monperrus 2024]. In this paper, we propose a novel approach to fault localization, called project-aware fault localization (PAFL).

Motivation. Our motivation is that a software project has its own difficulties and suffers from recurring faults associated with those project-specific difficulties. For example, Fig. 1 shows the distribution of fault patterns in two real-world projects. The *exiv2* project is a C++ library for reading, writing, and modifying image metadata, while *cppcheck* is a static analyzer for C/C++ programs. The *exiv2* project has a unique challenge of validating input metadata due to numerous variations of image metadata. Consequently, 70% of its faults are related to incorrect validation of input metadata. The *cppcheck* project, on the other hand, suffers from incorrect pattern matching for input tokens as the C++ language has various syntactic features [Padioleau 2009], accounting for 73% of the total faults.

This observation motivates our project-aware fault localization approach. Our hypothesis is as follows:

> *Leveraging project-specific fault patterns appeared in past versions would be beneficial for localizing faults in the future versions of the project.*

For instance, when localizing faults in the *exiv2* project, statements related to validating input metadata could be considered more suspicious than others. In contrast, in the *cppcheck* project, statements related to token pattern matching could be considered more suspicious.

Existing Approaches. To our knowledge, no existing fault localization techniques take project-specific fault patterns into account. Instead, existing techniques are designed with the assumption that faults are uniformly distributed across all projects. For example, spectrum-based fault localization techniques (SBFL) rely on pre-fixed formulas regardless of the given project [Abreu et al. 2006; Jones et al. 2002; Xie et al. 2013]. Existing machine learning-based approaches [B. Le et al. 2016; Lou et al. 2021; Meng et al. 2022; Sohn and Yoo 2017; Yang et al. 2020] also aim to learn a universal, project-unaware model; the model trained on one project is used for localizing faults in other projects.

Project-Aware Fault Localization. To address this limitation, PAFL enables existing fault localizers to leverage project-specific fault patterns. PAFL is a post-processing technique that treats baseline fault localizers as black-boxes and enhances them by updating their suspiciousness scores. Thus, PAFL is generally applicable to a wide range of baseline fault localizers, e.g., SBFL and DLFL.

PAFL is based on two key ideas. First, we design a domain-specific fault pattern-description language. An instance in our language, called crossword, describes a fault statement pattern and how the suspiciousness of a statement described by the crossword will be updated. If a statement is described by a crossword (i.e., the statement has the fault pattern), PAFL increases the suspiciousness of the statement. Second, we develop an algorithm that synthesizes project-specific crosswords

```
...
uint32_t resrcLength = getULong(buf, bigEndian);
+ enforce(resrcLength < io_->size(), ...);
while (resrcLength > 0)
...
```

(a) Fault and corresponding fix in *exiv2*#1.

```
...
subBox.length = getLong(...);
subBox.type = getLong(..., bigEndian);
- if (subBox.length > io_->size() - io_->tell()){
+ if (subBox.length < sizeof(box) || ...) {
  throw Error(kerCorruptedMetadata);}
```

(b) Fault and corresponding fix in *exiv2*#5.

Fig. 2. Faults appeared in two different versions of *exiv2* project.

from past buggy versions of the target project. The algorithm is specifically designed to efficiently synthesize high-quality crosswords from past buggy versions that can effectively improve the suspiciousness ranking of the fault statements in past versions.

We evaluated PAFL on seven representative baseline fault localization techniques (three SBFL, three DLFL, and one LLM-based FL) using 12 real-world C/C++ and Python projects with 224 bugs. The results show that PAFL effectively, robustly, and efficiently improves the performance of the baseline fault localizers. For example, PAFL enabled three baseline SBFL techniques, Ochiai [Abreu et al. 2006], DStar [Wong et al. 2012], and Barinel [Abreu et al. 2009], to rank 100%, 62.5%, and 160% more fault statements at the top-1 position (i.e., accurately ranked the fault statements as the most suspicious ones), respectively. PAFL also robustly improved the baselines; applying PAFL showed equal or better performance than the six baseline fault localizers for 93% of the 224 bugs. The overhead (i.e., training and score updating cost) of PAFL was negligible: the maximum overhead was less than one minute for all the versions of the projects.

Contributions. Our contributions are as follows:

- We present a novel technique, project-aware fault localization, that enables existing fault localization techniques to leverage project-specific fault patterns.
- We design a domain-specific language to describe fault patterns and an algorithm synthesizing project-specific fault patterns (i.e., crosswords) from past versions.
- We experimentally demonstrate that PAFL effectively, robustly, and efficiently improves the performance of various baseline fault localization techniques.

High-Level Idea of PAFL. The high-level idea of PAFL is to use suspicious tokens (e.g., variable name) to improve fault localization. Our observations also revealed that fault statements within a project are syntactically similar across different versions, allowing project-specific fault patterns to be described using suspicious tokens. PAFL mines these fault patterns (represented as crosswords) from past buggy versions and uses them to enhance fault localization in the latest version. The following section provides an overview of how PAFL leverages these suspicious tokens to improve fault localization.

2 Overview

In this section, we roughly illustrate our approach. We first introduce faults redundantly occurred in the two real-world projects *exiv2* and *cppcheck*, which motivates our project-aware fault localization approach. Then, we describe how our technique PAFL works.

Motivating Examples. Fig. 2 shows faults and corresponding fixes in two different versions of the *exiv2* project, a library and command-line utility to read, write, delete, and modify image metadata. The *exiv2* project includes complicated logic for processing input image metadata; hence, related

```
bool isTemporary(bool cpp, const Token* tok,...){
  ...
  if (Token::Match(tok->previous(), ...)){
    ...}
+ if (tok->isCast())
+  return false;
+ // Currying a function is unknown ...
+ if (Token::simpleMatch(tok, "(") && ...
+   return unknown;
  return true;}
```

(a) Fault and corresponding fix in *cppcheck*#10.

```
static bool isDeadTemporary(bool cpp, ...){
    if (!isTemporary(cpp, tok, library))
      return false;
-   if (expr && !precedes(...))
-     return false;
+   if (expr) {
+     if (!precedes(...))
+       return false;
+     const Token* parent = tok->astParent();
+     // Is in a for loop
+     if (astIsRHS(tok) && ...) {
+       const Token* braces = ...
+       if (precedes(braces, expr) && ...
+         return false;}}
    return true;}
```

(b) Fault and corresponding fix in *cppcheck*#18.

Fig. 3. Faults appeared in two different versions of *cppcheck* project.

faults have repeatedly occurred. For example, in the earlier version *exiv2*#1 in Fig. 2a, the original code missed a logic for processing the size of the input. A few versions later, *exiv2*#5 in Fig. 2b, the developer made a similar mistake in processing the size of the metadata. Such faults have frequently occurred during the development of the project, with 70% of the faults in *exiv2* being related to processing input metadata.

Fig. 3 shows the recurring faults and repairs in another software project, *cppcheck*, a static analysis tool for C/C++ code. The C++ language has numerous features; developing a comprehensive parser has been a significant challenge [Padioleau 2009]. It also has been a key challenge in the *cppcheck* project. For example, in the earlier version *cppcheck*#10 in Fig. 3a, the original code missed some conditions related to the variable '`tok`', which is responsible for parsing the tokens in a given C++ code. A few versions later, in *cppcheck*#18 in Fig. 3b, a similar fault occurred again. The original code also missed some conditions related to the variable '`tok`'. In the *cppcheck* project, 70% of the faults are related to parsing the tokens of input C++ code. We would like to note that the above challenges and fault patterns are project-specific. Other projects have different challenges and fault patterns. PAFL aims to leverage these project-specific fault patterns to improve fault localization.

Overview of PAFL. Fig. 4 shows the overall process of PAFL. Given a buggy project, PAFL first represents the program code as an aggregated AST, where each node in the tree represents a statement in the program (the details of aggregated AST are described in the next paragraph). Then, PAFL applies a baseline fault localization technique (e.g., OCHIAI [Abreu et al. 2006]) to get the initial suspiciousness score of each node (i.e., statement). After the initial suspiciousness scores are obtained, PAFL synthesizes fault patterns, namely crosswords, from the past buggy versions of the project. Intuitively, crosswords describe fault node patterns when a program is represented as an aggregated AST. Then, PAFL uses the mined crosswords to update the suspiciousness scores of the nodes. In the updating procedure, suspiciousness scores of nodes (i.e., statements) increase if they are described by the synthesized crosswords.

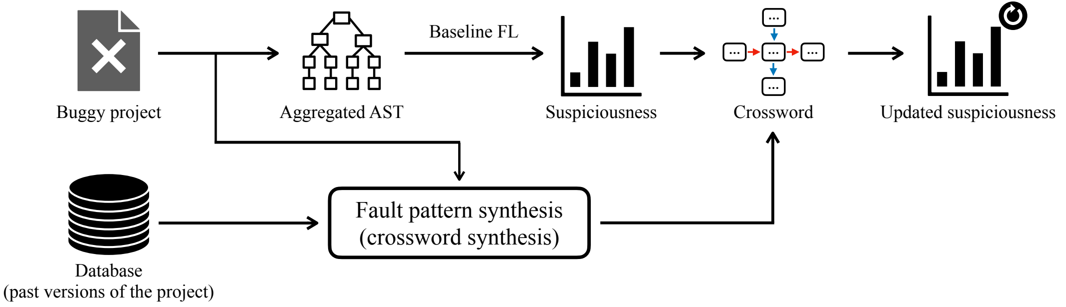
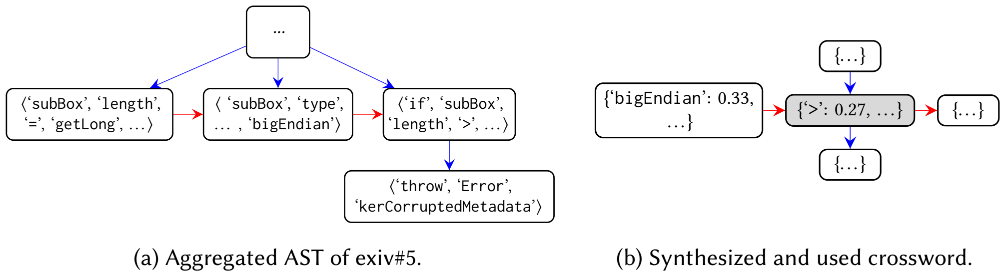


Fig. 4. Overall workflow of PAFL.



(a) Aggregated AST of exiv#5.

(b) Synthesized and used crossword.

Fig. 5. Aggregated AST of *exiv2*#5 (Fig. 2b) and the crossword mined from the earlier version *exiv2*#1 (Fig. 2a).

Aggregated AST. An aggregated AST is an abstract version of the full abstract syntax tree (AST). In an aggregated AST, each node represents a statement in the program, and the edges between the nodes represent the relationships between the statements. For example, Fig. 5a shows the aggregated AST of the buggy program *exiv2*#5 in Fig. 2b (i.e., the program before the fix). In the tree, the node ⟨'if', 'subBox', 'length', '>', ...⟩ represents the fault statement "`if (subBox.length > ...){` " in Fig. 2b. There are two types of edges in the aggregated AST. Red-colored edges describe the sequence relationships between the statements, and blue-colored edges describe the hierarchical (e.g., branch) relationships between the statements. For example, the statements "`subBox.length = getLong(...);`" and "`subBox.type = getLong(..., bigEndian);`" are sequentially connected in the program; the two nodes ⟨'subBox', 'length', '=', 'getLong', ...⟩ and ⟨ 'subBox', 'type', ... , 'bigEndian'⟩ are connected by a red-colored edge. The two nodes ⟨'throw', 'Error', 'kerCorruptedMetadata'⟩ and ⟨'if', 'subBox', 'length','>', ...⟩ are connected with a blue-colored edge because the former is a branch statement of the latter.

Crossword. Fig. 5b shows the crossword mined from the past version (*exiv2*#1 in Fig. 2a). A crossword is a graph with five nodes describing suspicious code patterns. Intuitively, the central node (gray-colored) and the adjacent nodes describe the patterns of fault statements and their adjacent statements, respectively. For instance, the crossword in Fig. 5b describes that fault statements usually include the token '>' and previous statements of fault statements include the variable '`bigEndian`'. The centered node {'>': 0.27, ...} increases the suspiciousness score by 0.27 if the statement includes the '>' token. The node {'bigEndian': 0.33, ...} increases the suspiciousness by 0.33 if a previous statement includes the '`bigEndian`' variable. For instance, the suspiciousness score

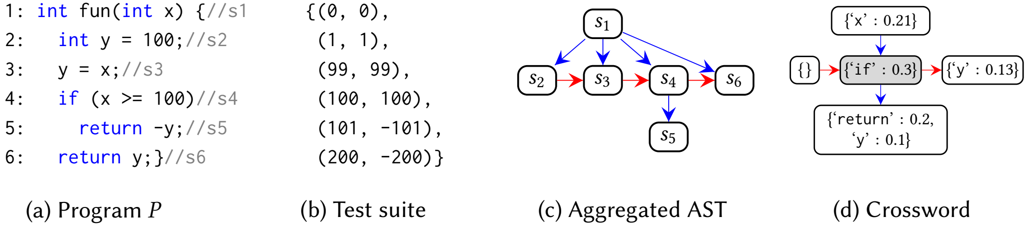


Fig. 6. Running Example

of the fault statement in Fig. 2b is increased by 0.6 as the statement "`if (subBox.length > ...`" (described by the node ⟨'`if`', '`subBox`', '`length`', '>', ...⟩ in Fig. 5a) includes the '>' token and the previous statement "`subBox.type = getLong(..., bigEndian);`" (⟨ '`subBox`', '`type`', ... , '`bigEndian`'⟩ in Fig. 5a) includes the '`bigEndian`' variable. In our evaluation, the synthesized crossword significantly improved the suspiciousness ranking of the fault statement in *exiv2*#5. The suspiciousness ranking of the fault statement in Fig. 2b increased from 38 to 1 when OCHIAI [Abreu et al. 2006] was used as the baseline fault localization technique.

We would like to note that the crossword describes a project-specific fault pattern using the '`bigEndian`' variable (i.e., token), which indicates the byte order of the header of the input metadata in the *exiv2* project. Applying the crossword to other projects may not be effective because the '`bigEndian`' is not a common variable in other projects.

3 Project-Aware Fault Localization

In this section, we describe our technique, PAFL, in detail using Fig. 6 as a running example.

Running Example. Fig. 6a shows an example buggy program. Fig. 6b presents a test suite, a collection of input-output values the program should satisfy. For instance, test $(1, 1)$ indicates that the program should return 1 when given input 1. The program does not satisfy the test case $(100, 100)$, as it returns -100 when it takes 100 as input. The fault lies in the token '>=' in line 4, which needs to be replaced with '>'.

3.1 Problem Definition

Program. We represent a program P as a sequence of statements, where each statement is a pair consisting of a label (i.e., line number) and a sequence of lexical tokens:

$$P \in \mathit{Program} = \mathit{Stmt}^*$$
$$s \in \mathit{Stmt} = \mathit{Label} \times \mathit{Token}^*$$
$$t \in \mathit{Token} = \mathit{Char}^*$$

For example, the example program in Fig. 6a is represented by the sequence $\langle s_1, s_2, s_3, s_4, s_5, s_6 \rangle$ with

$$s_1 = (1, \langle \texttt{int}, \texttt{fun}, \texttt{int}, \texttt{x} \rangle) \quad s_2 = (2, \langle \texttt{int}, \texttt{y}, =, \texttt{100} \rangle) \quad s_3 = (3, \langle \texttt{y}, =, \texttt{x} \rangle)$$
$$s_4 = (4, \langle \texttt{if}, \texttt{x}, >=, \texttt{100} \rangle) \quad s_5 = (5, \langle \texttt{return}, -, \texttt{y} \rangle) \quad s_6 = (6, \langle \texttt{return}, \texttt{y} \rangle)$$

Let $\mathit{tokens} : 2^{\mathit{Stmt}} \longrightarrow 2^{\mathit{Token}}$ be the function that extracts tokens from a set of statements: $\mathit{tokens}(S) = \bigcup_{(_,T) \in S} T$.

We represent a program as an aggregated AST. For instance, Fig. 6c shows the tree representation of the program in Fig. 6a. In this representation, we assume the following functions:

$$succ, pred, parent, child \in Program \times 2^{Stmt} \to 2^{Stmt}$$

which respectively retrieve the successors, predecessors, parents, and children of given statements, e.g., $succ(P, \{s_2\}) = \{s_3\}$, $pred(P, \{s_3\}) = \{s_2\}$, $parent(P, \{s_2\}) = \{s_1\}$, $child(P, \{s_1\}) = \{s_2, s_3, s_4, s_6\}$.

In the remainder of this section, when S is a singleton set, i.e., $S = \{s\}$, we simplify the notation for the functions *tokens*, *succ*, *pred*, *parent*, and *child* by omitting the braces. For example, we write $tokens(s)$ instead of $tokens(\{s\})$.

Execution. Let $[\![-]\!]$ be the instrumented program execution:

$$[\![P]\!] : In \to Out \times 2^{Stmt}$$

where In and Out denote the program's input and output domains, respectively. Note that $[\![P]\!]$ produces not only the output but also a set of statements executed by the input.

A test is a pair $(i, o) \in In \times Out$ of input and output values, and a test suite $T \in TestSuite = 2^{In \times Out}$ is a set of tests. Given a program P, we can classify tests in T into passing (T^P_{pass}) and failing (T^P_{fail}) tests:

$$T = T^P_{pass} \uplus T^P_{fail}$$
$$T^P_{pass} = \{(i, o) \in T \mid fst([\![P]\!](i)) = o\}$$
$$T^P_{fail} = \{(i, o) \in T \mid fst([\![P]\!](i)) \neq o\}$$

We say a program P is buggy w.r.t. a test suite T iff $T^P_{fail} \neq \emptyset$. Let $CovStmts(P, i)$ be the statements covered by $[\![P]\!](i)$:

$$CovStmts(P, i) = snd([\![P]\!](i))$$

In this work, we use statement-level fault localization. In the statement-level fault localization, faults represent statements that make the program fail test cases, and fixing the statements makes the program pass the test cases. If the fix only adds new statements, the previous and after statements of the added statements are considered fault statements [Pearson et al. 2017].

Baseline Fault Localizer. PAFL is designed to be used on top of a wide range of baseline fault localizers. We assume that a baseline fault localizer fl of the following type is given:

$$fl \in FL = Program \times TestSuite \to (Stmt \to \mathbb{R}).$$

Given a program P and a test suite T, $fl(P, T)$ maps statements to their suspiciousness scores. For example, Ochiai [Abreu et al. 2006] is defined as follows:

$$Ochiai(P, T) = \lambda s \in Stmt. \frac{N_f}{\sqrt{|T^P_{fail}| \times (N_f + N_p)}}$$

where N_f and N_p denote the numbers of failing and passing tests that cover statement s:

$$N_f = \sum_{(i,_) \in T^P_{fail}} \mathbb{1}_{s \in CovStmts(P,i)}, \quad N_p = \sum_{(i,_) \in T^P_{pass}} \mathbb{1}_{s \in CovStmts(P,i)}.$$

Goal. PAFL uses a set $V \subseteq Version$ of buggy program versions to transform a fault localizer into an enhanced one:

$$PAFL : FL \times 2^{Version} \to FL.$$

A buggy program version is defined as a tuple of buggy program, test suite, and ground truth:

$$Version = Program \times TestSuite \times GroundTruth.$$

where $GroundTruth = 2^{Stmt}$ denotes the set of statements modified by developers to fix the bug.

For an unseen buggy program P with test suite T and the ground truth G where the past buggy program versions are $V \subseteq Version$, PAFL aims to rank the fault statements higher than the baseline fault localizer fl:

$$FR(PAFL(fl, V)(P, T), G, P) \leq FR(fl(P, T), G, P)$$

where FR (First Ranking) [Xie et al. 2022] denotes the best suspiciousness ranking of the fault statements in G:

$$FR(f, G, P) = \min_{s \in G} Rank(f, s, P)$$

and $Rank(f, s, P)$ denotes the ranking of the statement s in program P when scoring function f is used:

$$Rank(f, s, P) = \sum_{s' \in P} \mathbb{1}_{f(s') \geq f(s)}.$$

That is, an enhanced fault localizer enables developers to face fault statements earlier. Note that we can use other metrics, such as AR (Average Ranking) [Xie et al. 2022], instead of FR. In this paper, we use FR, one of the most popular metrics used in prior works [Li et al. 2021; Xie et al. 2022].

3.2 Crosswords

Our key idea to solve the problem is to introduce and use a simple domain-specific language, *FPL* (*Fault Pattern-Description Language*), to describe various fault patterns in buggy program versions. PAFL updates the suspiciousness scores of each statement (i.e., node) using suspicious tokens (mined from the past buggy versions). If a statement is related to a suspicious token, the suspiciousness score increases. PAFL updates the suspiciousness scores by adding the maximum suspiciousness score of the tokens in the statement and its adjacent statements.

Intuitively, an instance of our domain-specific language $c \in FPL$, called crossword, describes a fault pattern using suspicious tokens and how they affect the suspiciousness scores of statements. A crossword maps each token to a suspiciousness score. If a statement contains these tokens, the suspiciousness score of the statement is increased with the mapped scores of the tokens. The following defines the syntax and semantics of our fault pattern description language.

Syntax. A crossword $c \in FPL$ is a tuple of five nodes:

$$FPL = Node \times Node \times Node \times Node \times Node.$$

Given a crossword $c = (n_1, n_2, n_3, n_4, n_5)$, we refer to n_1 as the center, n_2 the predecessor, n_3 the successor, n_4 the parent, and n_5 the child. A crossword can be visualized as shown in Fig. 6d, where:

- n_1 corresponds to the centered (gray-colored) node,
- n_2 and n_3 correspond to the predecessor and successor nodes of n_1, connected with red-colored edges, and
- n_4 and n_5 correspond to the parent and child nodes of n_1, connected with blue-colored edges.

A node contains a map from tokens to real numbers (i.e., suspiciousness scores of tokens):

$$Node = Token \rightarrow \mathbb{R}.$$

For example, node n_1 = {'if' : 0.3} in Fig. 6d maps the token '`if`' to 0.3, and node n_5 = {'return' : 0.2, 'y' : 0.1} maps tokens '`return`' and '`y`' to 0.2 and 0.1, respectively.

Semantics. We use crosswords to update the suspiciousness scores computed by the baseline fault localizer. Suppose that s is a statement in program P and $r \in \mathbb{R}$ is the suspiciousness score of s

computed by the baseline fault localizer. Applying $c = (n_1, n_2, n_3, n_4, n_5)$ to statement s, denoted $c(r, s, P)$, computes a new score as follows:

$$c(r,s,P) = r + \max_{t \in tokens(\{s\})} n_1(t) + \max_{t \in tokens(pred(P,s))} n_2(t) + \max_{t \in tokens(succ(P,s))} n_3(t) + \max_{t \in tokens(parent(P,s))} n_4(t) + \max_{t \in tokens(child(P,s))} n_5(t).$$

That is, a crossword increases the suspiciousness score of a statement if the statement or its adjacent statements contain tokens that are mapped in the corresponding nodes.

For example, the crossword in Fig. 6d increases the suspiciousness score of statement s_1 in Fig. 6a by 0.2 because its child statement s_6 contains the token '`return`' mapped to 0.2 in n_5. The suspiciousness score of statement s_2 is increased by 0.34 because its successor statement s_3 contains the token '`y`' and the parent statement s_1 contains the token '`x`', which are mapped to 0.13 and 0.21, respectively. Similarly, the suspiciousness scores of statements s_3 and s_6 are increased by 0.21 because their parent s_1 contains the token '`x`', which is mapped to 0.21 in n_4. The suspiciousness score of statement s_4 increases the most by the crossword. The score is increased by 0.84 as the statement s_4 contains the token '`if`', s_6 contains the token '`y`', s_1 contains the token '`x`', and s_5 contains the tokens '`return`', which are mapped to 0.3, 0.13, 0.21, and 0.2 in the nodes n_1, n_3, n_4, and n_5, respectively. The suspiciousness score of statement s_5 is increased by 0.21 because its parent statement s_4 contains the token '`x`' mapped to 0.21 in n_4.

If Ochiai is used as the baseline fault localizer, the suspiciousness scores of statements s_1, s_2, s_3, s_4, s_5, and s_6 are updated as follows (the column corresponding to the fault line s_4 is gray-colored, and the highest score is highlighted in bold):

	s_1	s_2	s_3	s_4	s_5	s_6
Before	0.44	0.44	0.44	0.44	**0.57**	0.0
After	0.64	0.78	0.65	**1.28**	0.78	0.21

Before updating the suspiciousness scores, s_4 has the second-highest score of 0.44, while s_5 has the highest score of 0.57. After the update, s_4 receives the highest score of 1.28; we can now identify the fault line as the top-1 suspicious statement.

We would like to note that formulas for adjusting the suspiciousness score are a design choice. For example, we can incorporate the number of tokens in each statement into the formula (e.g., statements with more tokens are considered more suspicious). In this work, we adopt a simple and straightforward approach that adds the maximum suspiciousness score of the tokens in a statement and its adjacent statements. Exploring more sophisticated formulas could enhance performance, making it an interesting avenue for future work.

Extension of Our Language. Our fault pattern-description language *FPL* can be extended in several ways to enhance its expressiveness. For example, one possible extension is to define a generalized language, *ExtendedFPL*, as follows:

$$ExtendedFPL = Node \times Node^* \times Node^* \times Node^* \times Node^*.$$

In the above generalized language, a crossword $(n_1, \overline{n}_2, \overline{n}_3, \overline{n}_4, \overline{n}_5) \in ExtendedFPL$ ($\overline{n}$ denotes a sequence of nodes) can have multiple predecessor, successor, ancestor, and descendant nodes. For example, $\overline{n}_4 = \langle n'_1, n'_2, \ldots, n'_p \rangle$ denotes ancestor nodes where the node n'_1 is a parent node of the center node n_1, the node n'_2 is a parent node of the node n'_1, and so on. Thus, the original language *Crossword* can be seen as a special case of *ExtendedFPL*, where each sequence has only one node. Additionally, *ExtendedFPL* can be further extended by considering the successor and predecessor (resp., parent and child) nodes of the parent and child nodes (resp., successor and predecessor).

Algorithm 1 PAFL

Input: Baseline fault localizer $fl \in FL$, buggy program versions $V = \{(P_1, T_1, G_1), \ldots, (P_n, T_n, G_n)\}$
Output: Enhanced fault localizer $fl' \in FL$

```
1:  procedure PAFL(fl, V)
2:      Π ←MineFaultPatterns(fl, V)
3:      fl′ ←EnhanceFL(fl, Π)
4:      return fl′
5:  end procedure
6:
7:  procedure EnhanceFL(fl, Π)
8:      procedure fl′(P, T)
9:          ℰ ← GetEmbedding(P, T)
10:         C ← CollectCrosswords(ℰ, Π)
11:         c ← MergeCrosswords(C)
12:         return λs ∈ P. c(fl(P, T)(s), s, P)
13:     end procedure
14:     return fl′
15: end procedure
```

However, using a more expressive pattern-description language does not necessarily improve fault localization performance. For example, we experimentally checked that using the extended language *ExtendedFPL* often resulted in worse performance than the original language *FPL*. The detailed results are presented and discussed in Section 4.3.

3.3 Project-Aware Fault Localization

Algorithm 1 presents our algorithm for project-aware fault localization (PAFL). It takes as input a baseline fault localizer $fl \in FL$ and a set $V \subseteq \mathit{Version}$ of buggy program versions:

$$V = \{(P_1, T_1, G_1), (P_2, T_2, G_2), ..., (P_n, T_n, G_n)\}.$$

As an output, it produces an enhanced fault localizer $fl' \in FL$.

Overall Procedure. The algorithm first mines (i.e., synthesizes) *fault patterns* from the given buggy versions. The MineFaultPatterns function in line 2 synthesizes a set of fault patterns $\Pi = \{(\mathcal{E}_1, c_1), \ldots, (\mathcal{E}_n, c_n)\}$, one for each program version:

$$(\mathcal{E}_i, c_i) \in \Pi \subseteq \mathit{FaultPattern} = \mathit{Embedding} \times \mathit{CrossWord}$$

where each fault pattern $(\mathcal{E}_i, c_i)$ consists of an *error embedding* $\mathcal{E}_i$ and a crossword c_i. The embedding $\mathcal{E}_i$ represents the fault in the ith version, i.e., (P_i, T_i, G_i), and c_i denotes the corresponding synthesized crossword. The method for synthesizing crosswords is explained in Section 3.4.

Next, the algorithm constructs a new fault localizer fl' by using EnhanceFL. The EnhanceFL procedure takes baseline fault localizer fl and fault patterns Π, and produces an enhanced localizer fl' following the steps in lines 8 through 14. The localizer fl' takes a program $P \in \mathit{Program}$ and a test suite $T \in \mathit{TestSuite}$ as input, and produces as output a map from statements to suspiciousness scores (line 12). In line 9, fl' gets the embedding $\mathcal{E}$ of the fault in program P using the *GetEmbedding* function. In line 10, the *CollectCrosswords* function produces a set $C \subseteq \mathit{CrossWord}$ of crosswords by collecting crosswords c_i whose embedding $\mathcal{E}_i$ matches $\mathcal{E}$. The collected crosswords C are merged into a single crossword c in line 11. In line 12, the suspiciousness scores of fl are updated by applying c to each statement s in P. The constructed fault localizer is returned in line 14.

Error Embedding. We represent faults in programs in terms of test execution. The embedding $\mathcal{E}$ is defined as a pair $(\mathcal{E}^{pass}, \mathcal{E}^{fail})$ of maps from tokens to natural numbers:

$$\mathcal{E} = (\mathcal{E}^{pass}, \mathcal{E}^{fail}) \in \mathit{Embedding} = (\mathit{Token} \rightarrow \mathbb{N})^2$$

where $\mathcal{E}^{pass} \in \mathit{Token} \rightarrow \mathbb{N}$ (resp., $\mathcal{E}^{fail} \in \mathit{Token} \rightarrow \mathbb{N}$) maps token t to the number of passing (resp., failing) tests that cover the token. The embedding is obtained using $\mathit{GetEmbedding} : \mathit{Program} \times \mathit{TestSuite} \rightarrow \mathit{Embedding}$, which is defined by

$$\mathit{GetEmbedding}(P, T) = (\mathcal{E}^{pass}, \mathcal{E}^{fail})$$

where $\mathcal{E}^{pass}$ and $\mathcal{E}^{fail}$ are:

$$\mathcal{E}^{pass} = \lambda t. \sum_{(i,_) \in T^P_{pass}} \sum_{s \in \mathit{CovStmts}(P,i)} \mathbb{1}_{t \in \mathit{tokens}(s)}$$

$$\mathcal{E}^{fail} = \lambda t. \sum_{(i,_) \in T^P_{fail}} \sum_{s \in \mathit{CovStmts}(P,i)} \mathbb{1}_{t \in \mathit{tokens}(s)}.$$

That is, $\mathcal{E}^{pass}$ and $\mathcal{E}^{fail}$ describe the number of passing and failing tests that cover each token in the program, respectively.

Collecting Synthesized Crosswords. Given an error embedding, the algorithm collects crosswords synthesized from past buggy versions with similar error embeddings. Function *CollectCrosswords* takes an embedding $\mathcal{E}$ and fault patterns $\Pi = \{(\mathcal{E}_1, c_1), \ldots, (\mathcal{E}_n, c_n)\}$, and collects crosswords c_i in Π such that $\mathcal{E}_i$ similar to $\mathcal{E}$:

$$\mathit{CollectCrosswords}(\mathcal{E}, \Pi) = \{c_i \mid (\mathcal{E}_i, c_i) \in \Pi, \mathcal{E} \approx \mathcal{E}_i\}\}$$

where $(\approx) \subseteq (\mathit{Token} \rightarrow \mathbb{N}) \times (\mathit{Token} \rightarrow \mathbb{N})$ denotes the similarity relation between embeddings. When $\mathcal{E} = (\mathcal{E}^{pass}, \mathcal{E}^{fail})$ and $\mathcal{E}_i = (\mathcal{E}_i^{pass}, \mathcal{E}_i^{fail})$, our definition of $\approx$ is as follows:

$$\mathcal{E} \approx \mathcal{E}_i \iff \|\mathcal{E}_i^{fail} - \mathcal{E}^{pass}\| \geq \|\mathcal{E}_i^{fail} - \mathcal{E}^{fail}\|$$

where $\|\mathcal{E}_1 - \mathcal{E}_2\|$ denotes the Euclidean distance between maps:

$$\|\mathcal{E}_1 - \mathcal{E}_2\| = \sqrt{\sum_{t \in \mathit{Token}} (\mathcal{E}_1(t) - \mathcal{E}_2(t))^2}.$$

The intuition is that two faults can be considered similar if failing tests cover similar tokens while passing tests cover different tokens. In other words, we consider two faults to be similar if the distance between the failure maps ($\|\mathcal{E}_i^{fail} - \mathcal{E}^{fail}\|$) is smaller than the distance between the failure and pass maps ($\|\mathcal{E}_i^{fail} - \mathcal{E}^{pass}\|$). We note that the similarity measure is also a design choice; other similarity measures can be used.

Merging Crosswords. Let $C = \{c_1, \ldots, c_k\}$ be a set of crosswords. Function $\mathit{MergeCrosswords} \in 2^{\mathit{CrossWord}} \rightarrow \mathit{CrossWord}$ merges C into a single crossword as follows:

$$\mathit{MergeCrosswords}(C) = \left(\lambda t. \sum_{i=1}^{k} n_1^i(t), \lambda t. \sum_{i=1}^{k} n_2^i(t), \lambda t. \sum_{i=1}^{k} n_3^i(t), \lambda t. \sum_{i=1}^{k} n_4^i(t), \lambda t. \sum_{i=1}^{k} n_5^i(t)\right)$$

where $c_i = (n_1^i, n_2^i, n_3^i, n_4^i, n_5^i)$. That is, crosswords mined from versions with similar faults are employed to update the suspiciousness scores in our approach.

Algorithm 2 Mining fault patterns

Input: Baseline fault localizer *fl*, buggy program versions $V = \{(P_1, T_1, G_1), \ldots, (P_n, T_n, G_n)\}$.
Output: Fault patterns $\Pi = \{(\mathcal{E}_1, c_1), \ldots, (\mathcal{E}_n, c_n)\}$

```
1:  procedure MineFaultPatterns(fl, V)
2:      for i = 1 to n do
3:          c_i ← (λt.0, λt.0, λt.0, λt.0, λt.0)
4:          E_i ← GetEmbedding(P_i, T_i)
5:          for j = 1 to i do
6:              if E_i ≈ E_j then
7:                  c_j ← Refine(c_j, (P_i, T_i, G_i), fl)
8:              end if
9:          end for
10:     end for
11:     return {(E_1, c_1), ..., (E_n, c_n)}
12: end procedure
```

3.4 Mining Fault Patterns

Now, we describe the MineFaultPatterns procedure, which synthesizes crosswords from past buggy versions of the target project.

Objective. The goal of the procedure is to produce fault patterns $\Pi = \{(\mathcal{E}_1, c_1), \ldots, (\mathcal{E}_n, c_n)\}$ from given buggy program versions $V = \{(P_1, T_1, G_1), \ldots, (P_n, T_n, G_n)\}$ and baseline fault localizer *fl*. The objective is to find Π that maximally enhances the performance of *fl* over V, i.e.,

$$\text{Find } \Pi \text{ that minimizes } \sum_{i=1}^{n} FR(fl'(P_i, T_i), G_i, P_i). \tag{1}$$

where $fl' = \textsc{EnhanceFL}(fl, \Pi)$ denotes the fault localizer enhanced by Π (Algorithm 1).

Overall Mining Algorithm. Algorithm 2 presents the overall fault pattern mining procedure. It takes as input a baseline fault localizer *fl* and buggy program versions $V = \{(P_1, T_1, G_1), \ldots, (P_n, T_n, G_n)\}$ where we assume that (P_i, T_i, G_i) denotes the ith buggy program version. As an output, the algorithm returns mined fault patterns $\Pi = \{(\mathcal{E}_1, c_1), \ldots, (\mathcal{E}_n, c_n)\}$.

In lines 2 through 10, the algorithm iteratively synthesizes and updates crosswords using the ith buggy program version. In line 3, the algorithm first initializes a new crossword c_i. Then, it gets the error embedding $\mathcal{E}_i$ of the ith version (line 4). In lines 5 through 9, the algorithm updates the crosswords $\{c_1, \ldots, c_i\}$ using the ith version. In each iteration, the algorithm updates the crossword c_j using Refine if the corresponding error embedding $\mathcal{E}_j$ is similar to the error embedding $\mathcal{E}_i$ (i.e., $\mathcal{E}_i \approx \mathcal{E}_j$). That is, crosswords that will be used when localizing faults in the ith program version are updated. The function $\textsc{Refine}(c_j, (P_i, T_i, G_i), fl)$ aims to refine the crossword c_j; the refined crossword c_j effectively localizes faults in the ith program version. We expect that the refined crosswords $\{c_1, \ldots, c_n\}$ would minimize (1). Upon termination, the algorithm returns the mined fault patterns.

Refining a Crossword. Our crossword-refinement procedure Refine refines the given crossword c to effectively localize faults in the given buggy program version (P, T, G). It mutates each node in the input crossword using various tokens (in the buggy version of the project) and incorporates the effects of these mutations into the refined crossword. Algorithm 3 presents the crossword-refining procedure Refine. It takes as input a crossword $c = (n_1, n_2, n_3, n_4, n_5)$, a buggy program version

Algorithm 3 Refining a crossword

Input: Crossword $c = (n_1, n_2, n_3, n_4, n_5)$, buggy program version (P, T, G), fault localizer fl
Output: Updated crossword $(n_1'', n_2'', n_3'', n_4'', n_5'')$

1: **procedure** Refine(c, (P, T, G), fl)
2: $(n_1'', n_2'', n_3'', n_4'', n_5'') \leftarrow c$
3: **for each** $i \in \{1, 2, 3, 4, 5\}$ **do**
4: $W \leftarrow TokensRelatedToFaults(P, G, i)$
5: $W \leftarrow W \cup \{t \mid n_i(t) > 0\}$
6: **for each** $t \in W$ **do**
7: $c' \leftarrow Mutate(c, i, t)$
8: $r \leftarrow FR(\lambda s.\ c(fl(P, T)(s), G, P))$
9: $r' \leftarrow FR(\lambda s.\ c'(fl(P, T)(s), G, P))$
10: $n_i'' \leftarrow UpdateNode(n_i'', t, r, r')$
11: **end for**
12: **end for**
13: **return** $(n_1'', n_2'', n_3'', n_4'', n_5'')$
14: **end procedure**

(P, T, G), and a baseline fault localizer fl. As output, it returns a refined crossword. In line 2, the algorithm initializes a crossword $(n_1'', n_2'', n_3'', n_4'', n_5'')$ with c. In lines 3 through 12, the algorithm iteratively mutates a node n_i in the crossword $c = (n_1, n_2, n_3, n_4, n_5)$ and refines the node n_i''. In lines 4 and 5, the algorithm collects candidate tokens as W for the mutation:

$$TokensRelatedToFaults(P, G, i) = \begin{cases} tokens(G) & \text{if } i = 1 \\ tokens(pred(P, G)) & \text{if } i = 2 \\ tokens(succ(P, G)) & \text{if } i = 3 \\ tokens(parent(P, G)) & \text{if } i = 4 \\ tokens(child(P, G)) & \text{if } i = 5 \end{cases}$$

Intuitively, $TokensRelatedToFaults(P, G, i)$ collects promising tokens such that assigning a high score in n_i would improve the fault localization for the given buggy version. The algorithm also collects the tokens mapped in the node n_i for checking whether the tokens have negative effect in the localization.

In lines 6 through 11, the algorithm iteratively mutates a node n_i using a token $t \in W$, and updates the node n_i''. In line 8, the algorithm mutates the given crossword with $Mutate((n_1, n_2, n_3, n_4, n_5), i, t)$. It returns an updated crossword $(n_1', n_2', n_3', n_4', n_5')$ satisfying: for $j \in \{1, \ldots, 5\}$,

$$n_j' = \begin{cases} n_j & \text{if } i \neq j \\ \lambda t'.\text{if } t' = t \text{ then } 1 \text{ else } n_j(t) & \text{if } i = j. \end{cases}$$

Our algorithm is designed to assign a score in $[0, 1]$ to a token in crosswords; 1 is the highest score that can be assigned. After the mutation, the algorithm gets the performance of the original and mutated crosswords as r and r', respectively. Then, the algorithm reflects the effect of the mutation to the node n_i'' using the function $UpdateNode(n_i'', t, r, r')$ (line 10). It returns a refined node as follows:

- If $r' \leq r$ (the performance is improved), it returns:

$$\lambda t'.\ \text{if } t' = t \text{ then } n_i''(t) + (1 - n_i''(t)) \times (1 - \frac{r'}{r}) \text{ else } n_i''(t').$$

- Otherwise (the performance is unimproved), it returns:

$$\lambda t'. \text{ if } t' = t \text{ then } n_i''(t) - (n_i''(t)) \times (1 - \frac{r}{r'}) \text{ else } n_i''(t').$$

That is, if the mutation is beneficial (resp., detrimental), the score mapped to the token t increases toward 1 (resp., decreases toward 0). Note that the updated score is closer to 1 (resp., 0) if the mutation is more effective. After the iterations, the algorithm returns the updated crossword $(n_1'', n_2'', n_3'', n_4'', n_5'')$ in line 13.

3.5 Running Example of Refining a Crossword

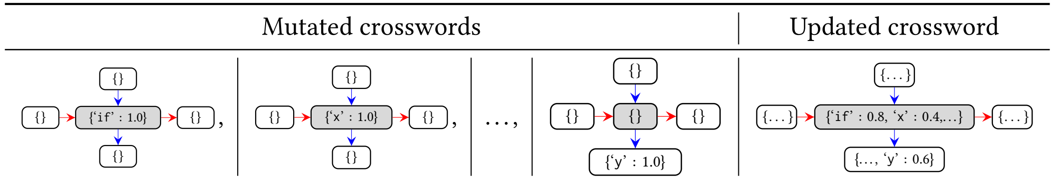
Suppose the given input crossword c is $(\lambda t.0, \lambda t.0, \lambda t.0, \lambda t.0, \lambda t.0)$, the baseline fault localizer is Ochiai, and the buggy program version (P, T, G) corresponds to the running example in Fig. 6 as follows.

$$\begin{aligned} P &= \langle(1, \langle \texttt{int}, \texttt{fun}, \texttt{int}, \texttt{x}\rangle), (2, \langle \texttt{int}, \texttt{y}, \texttt{=}, \texttt{100}\rangle), \ldots, (5, \langle \texttt{return}, -, \texttt{y}\rangle), (6, \langle \texttt{return}, \texttt{y}\rangle)\rangle \\ T &= \{(0,0), (1,1), (99,99), (100,100), (101,-101), (200,-200)\} \\ G &= \{(4, \langle \texttt{if}, \texttt{x}, \texttt{>=}, \texttt{100}\rangle)\} \end{aligned}$$

The fault-related tokens are then collected using $\mathit{TokensRelatedToFaults}(P, G, i)$, as follows:

$$\mathit{TokensRelatedToFaults}(P, G, i) = \begin{cases} \{\texttt{if}, \texttt{x}, \texttt{>=}, \texttt{100}\} & \text{if } i = 1 \\ \{\texttt{y}, \texttt{=}, \texttt{x}\} & \text{if } i = 2 \\ \{\texttt{return}, \texttt{y}\} & \text{if } i = 3 \\ \{\texttt{int}, \texttt{fun}, \texttt{x}\} & \text{if } i = 4 \\ \{\texttt{return}, -, \texttt{y}\} & \text{if } i = 5 \end{cases}$$

From the collected tokens, the algorithm enumerates 15 mutated crosswords (# of collected tokens above) and refines the input crossword c using these mutations as follows:



The first and second mutations increase the suspiciousness ranking of the faulty statement (i.e., $(4, \langle \texttt{if}, \texttt{x}, \texttt{>=}, \texttt{100}\rangle)$) from 5 to 1 and 3[1], respectively. In the updated crossword, the tokens `if'` and `x'` in {if' : 0.8, x' : 0.4,...} are assigned values of 0.8 $(1 - \frac{1}{5})$ and 0.4 $(1 - \frac{3}{5})$, respectively.

4 Evaluation

In this section, we experimentally evaluate PAFL. Our evaluation aims to answer the following research questions:

- **(RQ1) Performance of PAFL:** Does PAFL effectively, robustly, and efficiently improve the performance of baseline fault localizers?
- **(RQ2) Comparison with State-of-the-Art:** How does PAFL perform compared to existing state-of-the-art fault localizer enhancing techniques?
- **(RQ3) Effect of language extension:** How does the extension of fault pattern-description language affect the performance of PAFL?
- **(RQ4) Learned Insight:** Can the used crosswords (i.e., fault patterns) provide meaningful insights to the developers?

[1]We use the max tie-breaker, which assigns the worst ranking to tied statements (i.e., those with the same suspiciousness scores). Details of the max tie-breaker are provided in Section 4.

Table 1. Statistics of the benchmark projects.

Language	Project	#Versions	LOC	Test LOC	#Tests	#Fault stmts
C	*libchewing*	8	106.0K	3.6K	18	3.5
	libxml2	7	231.0K	33.5K	41	6.6
	openssl	26	947.8K	102.3K	226	8.3
C++	*cpp-peglib*	10	15.4K	1.7K	134	6.0
	cppcheck	30	274.9K	44.6K	92	5.9
	exiv2	20	83.4K	17.5K	6	4.3
	proj	28	215.3K	42.5K	54	8.4
	yaml-cpp	10	76.8K	19.6K	15	2.0
Python	*thefuck*	31	9.8K	2.9K	1,649	3.2
	fastapi	15	23.4K	1.7K	432	3.5
	spacy	7	97.2K	6.2K	1,711	2.0
	YouTube-dl	32	119.8K	15.9K	223	2.9

Benchmarks. We implemented PAFL for C/C++ and Python and evaluated it on 12 real-world projects. For C/C++, we used eight benchmark projects from BugsC++ [An et al. 2023], a collection of C/C++ projects with test cases. Among the benchmark projects, we collected those with at least seven buggy versions and reproducible bugs. For Python, we collected four projects from BugsInPy [Widyasari et al. 2020] using the same selection criteria. We would like to note that real-world projects usually satisfy the criteria of having at least seven buggy versions with reproducible bugs. The majority of modern real-world projects are developed with version control systems such as Git, and enough number of (e.g., more than seven) bugs and fixes have usually been accumulated during their development. Also, the buggy and fix versions can be easily collected using commit messages (e.g., whether the commit messages include 'fix').

Table 1 shows detailed information about the 12 projects. The column "Project" lists the project names, and the column "#Versions" shows the number of buggy versions in the projects. For example, PAFL was evaluated on 224 buggy versions in total. The columns "LOC," "#Tests," and "Test LOC" represent the number of lines, test cases, and lines covered by the test case that covers the most lines, respectively, in the latest version of each project. The column "#Fault stmts" shows the average number of ground-truth fault statements across the buggy versions.

Setup. When evaluating PAFL for each project, we first sorted the project versions chronologically and then sequentially evaluated PAFL on the sorted versions. That is when evaluating PAFL on the nth version of a project, we used the past $n-1$ versions $V = \{(P_1, T_1, G_1), \ldots, (P_{n-1}, T_{n-1}, G_{n-1})\}$ as a training data to mine the fault patterns $\Pi = \{(\mathcal{E}_1, c_1), \ldots, (\mathcal{E}_{n-1}, c_{n-1})\}$. All experiments were conducted on a 64-bit Linux (Ubuntu 18.04.5) server with 2.2 GHz Intel(R) Xeon CPUs, 128 GB RAM, and an NVIDIA RTX A6000 GPU with 48 GB of memory.

Baseline Fault Localizers. We used seven baseline fault localization techniques used in recent work [Xie et al. 2022]: Ochiai [Abreu et al. 2006], DStar [Wong et al. 2012], Barinel [Abreu et al. 2009], MLP-FL [Chen et al. 2016], CNN-FL [Zhang et al. 2019], RNN-FL [Zhang et al. 2021], and LLMAO [Yang et al. 2024]. Ochiai, DStar, and Barinel are well-known spectrum-based fault localization (SBFL) techniques. MLP-FL, CNN-FL, and RNN-FL are deep learning-based fault localization (DLFL) techniques. LLMAO is a state-of-the-art LLM-based fault localization technique. In our experiments on DLFL techniques, we used the artifact of Xie et al. [2022], which

provides the implementations of the three baseline DLFL techniques. When evaluating LLMAO, we used the artifact provided by the authors. Since PAFL is designed for statement-level fault localization, we excluded coarser-level (e.g., method-level) fault localizers such as DeepFL [Li et al. 2019], Fluccs [Sohn and Yoo 2017], TraPT [Li and Zhang 2017], ProFL [Lou et al. 2020], and BugPecker [Cao et al. 2020].

Metrics. We measure the performance using the following widely used metrics [Xie et al. 2022]:

- **Number of Top-K** [Kochhar et al. 2016]: The number of buggy versions where a faulty statement is included in the top-K suspicious statements.
- **MFR (Mean First Rank)** [Lou et al. 2020]: FR (First Rank) is the best ranking of faulty statements in a version. MFR is the mean value of FR over the versions.
- **MAR (Mean Average Rank)** [Lou et al. 2020]: AR (Average Rank) is the average ranking of faulty statements in a version. MAR is the mean value of AR over the versions.

Tiebreaking. When multiple statements have the same suspiciousness scores (i.e., tie), we used the *max* tie-breaker [Yu et al. 2008]. The *max* tie-breaker assigns the sum of the number of the tied statements and the number of statements with higher suspiciousness scores to the tied statements [Sohn and Yoo 2017] (i.e., assigning the worst ranking to the tied statements).

4.1 RQ1: Performance of PAFL

PAFL effectively, robustly, and efficiently improved the performance of the baseline fault localization techniques. Table 2, Table 3, and Table 4 show the evaluation results of PAFL on the SBFL, DLFL, and LLM-based FL techniques, respectively, for the 12 projects. In Table, for example, 2 the columns "Ochiai," "DStar," and "Barinel" present the performance of the three baseline SBFL techniques, and the columns "PAFL" show the performance of PAFL when the corresponding baselines were used. The columns "Delta" represent the relative improvement of PAFL compared to the baselines. For example, the "Delta" of the top-10 metric (higher is better) describes the increment in the metric. The "Delta" of the MFR and MAR metrics (lower is better) describes the decrease in the metrics. The best performance for each metric is highlighted in bold.

Effectiveness of PAFL on SBFL. Table 2 shows that PAFL effectively improved the baseline SBFL techniques for the 12 projects and three metrics. For example, applying PAFL improved the performance of the baselines for all the 12 projects in terms of the MFR and MAR metrics. In particular, in terms of the top-K metrics, PAFL significantly improved the SBFL techniques. For example, PAFL improved 100%, 62.5%, and 160% of the top-1 metric in Ochiai, DStar, and Barinel, respectively, in total. That is, PAFL enabled the localizers to provide the faulty statements at the first rank 62.5% ~ 160% more frequently. For the top-5 and top-10 metrics, PAFL improved the baseline SBFL techniques by an average of 83.4% and 52.4%, respectively. We would like to note that SBFL techniques are actively used in practice because of their simplicity, and Top-1 and Top-5 are the most important metrics from a practical perspective [Kochhar et al. 2016].

Effectiveness of PAFL on DLFL. Table 3 shows the evaluation results of PAFL on the DLFL techniques. In the literature, the baseline DLFL techniques had shown worse performance than the SBFL techniques in Java projects [Xie et al. 2022]; this was also the case in our evaluation. For example, the MFR of MLP-FL and Ochiai for the *cpp-peglib* project are 219.2 and 151.0 (i.e., Ochiai is 31.1% better than MLP-FL), respectively. In the table, the columns "MLP-FL," "CNN-FL," and "RNN-FL" present the performance of the three baseline DLFL techniques, and the columns "PAFL" show the performance when the corresponding baselines were used. As the DLFL techniques have randomness in their learning procedure, we ran each one five times and reported the average

Table 2. Performance of PAFL for the three baseline SBFL techniques across the 12 C/C++ and Python projects. The columns "Ochiai," "DStar," and "Barinel" present the performance of the baseline fault localizers. The columns "PAFL" present the performance of our technique when the corresponding baselines were used. The columns "Delta" present the relative improvement when PAFL was applied.

Project	Metric	Ochiai	PAFL	Delta	DStar	PAFL	Delta	Barinel	PAFL	Delta
cpp-peglib	MFR	151.0	**138.8**	8.1%	151.0	**138.0**	8.6%	153.3	**140.3**	8.5%
	MAR	208.6	**195.4**	6.3%	208.6	**194.6**	6.7%	210.9	**196.5**	6.8%
	Top-10	**3**	**3**	0.0%	**3**	**3**	0.0%	**3**	**3**	0.0%
cppcheck	MFR	2883.1	**2740.9**	4.9%	2883.1	**2690.1**	6.7%	2883.1	**2710.1**	6.0%
	MAR	3654.4	**3492.2**	4.4%	3654.4	**3380.1**	7.5%	3654.4	**3431.2**	6.1%
	Top-10	0	**2**	∞%	0	**2**	∞%	0	**2**	∞%
exiv2	MFR	607.1	**283.0**	53.4%	607.1	**283.1**	53.4%	580.3	**228.4**	60.6%
	MAR	673.4	**306.8**	54.4%	673.4	**307.1**	54.4%	646.6	**252.7**	60.9%
	Top-10	5	**10**	100.0%	5	**10**	100.0%	4	**9**	125.0%
libchewing	MFR	776.0	**690.5**	11.0%	776.0	**693.8**	10.6%	776.0	**693.8**	10.6%
	MAR	802.6	**716.7**	10.7%	802.6	**718.9**	10.4%	802.6	**718.8**	10.4%
	Top-10	**0**	**0**	0.0%	**0**	**0**	0.0%	**0**	**0**	0.0%
libxml2	MFR	322.7	**309.4**	4.1%	322.7	**309.6**	4.1%	322.7	**308.0**	4.6%
	MAR	709.1	**662.3**	6.6%	709.1	**627.3**	11.5%	709.1	**636.8**	10.2%
	Top-10	1	**2**	100.0%	1	**2**	100.0%	1	**2**	100.0%
proj	MFR	8741.6	**6937.8**	20.6%	8744.8	**6947.6**	20.6%	8832.8	**7036.5**	20.3%
	MAR	9175.9	**7983.0**	13.0%	9179.1	**7985.6**	13.0%	9267.1	**8072.3**	12.9%
	Top-10	**0**	**0**	0.0%	**0**	**0**	0.0%	**0**	**0**	0.0%
openssl	MFR	10514.2	**9296.8**	11.6%	10635.2	**8387.7**	21.1%	11471.1	**10142.5**	11.6%
	MAR	11979.0	**11039.8**	7.8%	12198.0	**10453.8**	14.3%	12616.0	**11688.3**	7.4%
	Top-10	**0**	**0**	0.0%	**0**	**0**	0.0%	**0**	**0**	0.0%
yaml-cpp	MFR	370.4	**335.8**	9.3%	370.4	**335.5**	9.4%	370.4	**335.8**	9.3%
	MAR	379.8	**345.3**	9.1%	379.8	**345.0**	9.2%	379.8	**345.3**	9.1%
	Top-10	**0**	**0**	0.0%	**0**	**0**	0.0%	**0**	**0**	0.0%
thefuck	MFR	16.0	**13.6**	15.1%	15.1	**12.9**	14.1%	18.7	**12.9**	31.0%
	MAR	19.5	**17.1**	12.2%	18.4	**16.3**	11.3%	22.8	**17.7**	22.5%
	Top-10	18	**20**	11.1%	**21**	**21**	0.0%	12	**20**	66.7%
fastapi	MFR	189.2	**185.1**	2.2%	196.5	**188.1**	4.2%	84.3	**80.3**	4.7%
	MAR	195.2	**191.5**	1.9%	201.6	**199.2**	1.2%	86.9	**83.2**	4.3%
	Top-10	**5**	**5**	0.0%	**5**	**5**	0.0%	**5**	**5**	0.0%
spacy	MFR	28.4	**17.3**	39.2%	28.4	**17.3**	39.2%	28.4	**17.3**	39.2%
	MAR	30.6	**25.0**	18.2%	30.6	**25.0**	18.2%	30.6	**25.0**	18.2%
	Top-10	3	**4**	33.3%	3	**4**	33.3%	3	**4**	33.3%
YouTube-dl	MFR	273.2	**229.1**	16.1%	1470.5	**1132.0**	23.0%	173.8	**117.7**	32.3%
	MAR	284.9	**244.3**	14.3%	1683.2	**1350.3**	19.8%	179.9	**131.2**	27.1%
	Top-10	0	**3**	∞%	0	**3**	∞%	0	**7**	∞%
total	MFR	2869.3	**2441.7**	14.9%	3055.1	**2459.8**	19.5%	2968.6	**2520.3**	15.1%
	MAR	3221.5	**2895.7**	10.1%	3447.3	**2970.3**	13.8%	3282.7	**2945.1**	10.3%
	Top-1	7	**14**	100.0%	8	**13**	62.5%	5	**13**	160.0%
	Top-5	22	**36**	63.6%	25	**40**	60.0%	15	**34**	126.7%
	Top-10	35	**49**	40.0%	38	**50**	31.6%	28	**52**	85.7%

Table 3. Performance of PAFL for the three baseline DLFL techniques across the 12 C/C++ and Python projects. The columns "MLP-FL," "CNN-FL," and "RNN-FL" present the performance of the baseline fault localizers. The columns "PAFL" present the performance of our technique when the corresponding baselines were used. The columns "Delta" present the relative improvement when PAFL was applied. OOM indicates that out-of-memory errors occurred in the baseline localizers; the corresponding results of PAFL are unavailable (i.e., N/A).

Project	Metric	MLP-FL	PAFL	Delta	CNN-FL	PAFL	Delta	RNN-FL	PAFL	Delta
cpp-peglib	MFR	219.2	**219.2**	0.0%	328.6	**300.3**	8.6%	220.2	**220.1**	0.0%
	MAR	319.6	**319.4**	0.1%	399.3	**385.7**	3.4%	320.9	**320.7**	0.1%
	Top-10	**0.2**	**0.2**	0.0%	**0.0**	**0.0**	0.0%	**0.8**	**0.8**	0.0%
cppcheck	MFR	3524.4	**3376.3**	4.2%	OOM	N/A	N/A	OOM	N/A	N/A
	MAR	5494.5	**5342.8**	2.8%	OOM	N/A	N/A	OOM	N/A	N/A
	Top-10	0.4	**0.6**	50.0%	OOM	N/A	N/A	OOM	N/A	N/A
exiv2	MFR	461.2	**433.2**	6.1%	OOM	N/A	N/A	OOM	N/A	N/A
	MAR	711.2	**678.2**	4.6%	OOM	N/A	N/A	OOM	N/A	N/A
	Top-10	1.2	**2.0**	66.7%	OOM	N/A	N/A	OOM	N/A	N/A
libchewing	MFR	640.7	**638.0**	0.4%	1053.0	**990.8**	5.9%	712.8	**709.5**	0.5%
	MAR	907.4	**904.9**	0.3%	1211.7	**1158.0**	4.4%	942.1	**939.1**	0.3%
	Top-10	**0.4**	**0.4**	0.0%	**0.0**	**0.0**	0.0%	**0.0**	**0.0**	0.0%
libxml2	MFR	539.7	**510.5**	5.4%	OOM	N/A	N/A	OOM	N/A	N/A
	MAR	1077.0	**1010.2**	6.2%	OOM	N/A	N/A	OOM	N/A	N/A
	Top-10	0.0	**0.2**	∞%	OOM	N/A	N/A	OOM	N/A	N/A
proj	MFR	4897.4	**4873.0**	0.5%	OOM	N/A	N/A	OOM	N/A	N/A
	MAR	7404.8	**7368.8**	0.5%	OOM	N/A	N/A	OOM	N/A	N/A
	Top-10	**0.8**	**0.8**	0.0%	OOM	N/A	N/A	OOM	N/A	N/A
openssl	MFR	OOM	N/A	N/A	OOM	N/A	N/A	OOM	N/A	N/A
	MAR	OOM	N/A	N/A	OOM	N/A	N/A	OOM	N/A	N/A
	Top-10	OOM	N/A	N/A	OOM	N/A	N/A	OOM	N/A	N/A
yaml-cpp	MFR	307.4	**305.6**	0.6%	OOM	N/A	N/A	OOM	N/A	N/A
	MAR	335.0	**333.7**	0.4%	OOM	N/A	N/A	OOM	N/A	N/A
	Top-10	0.8	**1.0**	25.0%	OOM	N/A	N/A	OOM	N/A	N/A
thefuck	MFR	28.9	**28.2**	2.3%	147.9	**120.2**	18.7%	33.4	**32.6**	2.5%
	MAR	39.5	**38.8**	1.9%	151.5	**124.1**	18.1%	42.7	**41.9**	2.1%
	Top-10	13.0	**13.2**	1.5%	0.0	**2.8**	∞%	10.6	**11.0**	3.8%
fastapi	MFR	186.1	**184.2**	1.0%	394.2	**362.1**	8.1%	128.9	**128.7**	0.1%
	MAR	257.7	**255.6**	0.8%	462.2	**443.1**	4.1%	193.7	**191.8**	1.0%
	Top-10	**2.0**	**2.0**	0.0%	0.0	**0.6**	∞%	**1.6**	**1.6**	0.0%
spacy	MFR	242.2	**242.1**	0.0%	672.2	**650.6**	3.2%	193.6	**193.6**	0.0%
	MAR	343.5	**343.2**	0.1%	756.1	**748.9**	0.9%	**256.9**	256.9	-0.0%
	Top-10	**0.8**	**0.8**	0.0%	0.2	**0.6**	200.0%	**0.6**	**0.6**	0.0%
YouTube-dl	MFR	2560.3	**2505.9**	2.1%	OOM	N/A	N/A	OOM	N/A	N/A
	MAR	3413.8	**3348.7**	1.9%	OOM	N/A	N/A	OOM	N/A	N/A
	Top-10	**0.2**	**0.2**	0.0%	OOM	N/A	N/A	OOM	N/A	N/A
total	MFR	1273.2	**1231.8**	3.3%	379.0	**347.1**	8.4%	172.2	**171.4**	0.5%
	MAR	1865.5	**1819.3**	2.5%	431.1	**406.5**	5.7%	236.2	**235.1**	0.5%
	Top-1	**6.4**	**6.4**	0.0%	0.2	**1.0**	400.0%	**0.0**	**0.0**	0.0%
	Top-5	12.4	**13.0**	4.8%	0.2	**2.6**	1200.0%	**8.2**	**8.2**	0.0%
	Top-10	19.0	**20.6**	8.4%	0.2	**4.0**	1900.0%	13.6	**14.0**	2.9%

Table 4. Performance of PAFL for the LLM-based fault localizer LLMAO.

Project	Metric	LLMAO	PAFL	Delta	Project	Metric	LLMAO	PAFL	Delta
cpp-peglib	MFR	320.9	**320.8**	0.0%	*yaml-cpp*	MFR	**357.6**	357.8	-0.1%
	MAR	412.1	**408.2**	0.9%		MAR	**377.9**	379.8	-0.5%
cppcheck	MFR	4060.7	**3668.3**	9.7%	*thefuck*	MFR	172.2	**111.9**	35.0%
	MAR	5486.5	**5178.3**	5.6%		MAR	262.4	**179.0**	31.8%
exiv2	MFR	1167.3	**1055.8**	9.6%	*fastapi*	MFR	246.2	**218.3**	11.3%
	MAR	1435.9	**1244.7**	13.3%		MAR	315.3	**286.8**	9.0%
libchewing	MFR	**507.1**	508.9	-0.3%	*spacy*	MFR	**519.7**	522.4	-0.5%
	MAR	787.5	**773.9**	1.7%		MAR	**561.6**	563.0	-0.2%
libxml22	MFR	758.4	**704.6**	7.1%	*YouTube-dl*	MFR	2812.9	**2451.0**	12.9%
	MAR	1447.2	**1265.5**	12.6%		MAR	3446.2	**2887.1**	16.2%
proj	MFR	6110.8	**5935.1**	2.9%	*total*	MFR	3112.7	**2941.8**	5.5%
	MAR	8411.8	**7698.0**	8.5%		MAR	4515.7	**4253.6**	5.8%
openssl	MFR	10082.6	**9884.5**	2.0%		Top-1	**1**	**1**	0.0%
	MAR	16587.6	**16457.9**	0.8%		Top-10	**11**	**11**	0.0%

results. In the table, OOM indicates that the baseline techniques failed the localization procedure due to out-of-memory errors[2].

PAFL also improved the baseline DLFL techniques. In terms of MFR and MAR, PAFL improved the baselines for all 21 cases (100%) and 20 out of 21 (95.2%) cases, respectively. In the Top-10 metric, PAFL showed equal or better performance for all three baselines and 12 projects. In Table 2 and Table 3, there are 72 cases (12 projects × 6 baselines) of the results, and 57 of them are available. In terms of the MFR and MAR metrics, PAFL improved the baselines in all 57 cases (100%) and 56 out of 57 cases (98.2%), respectively. For the top-10 metric, PAFL showed equal or better performance for all 57 cases (100%).

Effectiveness of PAFL on LLM-based FL. Table 4 shows the evaluation results of PAFL when the state-of-the-art LLM-based fault localizer LLMAO [Yang et al. 2024] was used as a baseline. When localizing a fault in a buggy version of a project, we trained LLMAO using the other buggy versions of the project, enabling the model to learn project-specific fault patterns. In our evaluation, the LLM-based approach also shows worse performance than the SBFL techniques. For instance, the MFR of LLMAO and Ochiai for the *cpp-peglib* project are 151.0 and 320.9 (i.e., Ochiai performed about 112.8% better than LLMAO), respectively. Overall, PAFL also improved the performance of LLMAO. In the *thefuck* project, PAFL improved the MFR and MAR scores by 35.0% and 31.8%, respectively. In two projects, *yaml-cpp*, and *spacy*, however, PAFL slightly degraded (e.g., -0.5%) the performance of LLMAO in terms of the MFR and MAR metrics. In total, PAFL improved the MFR and MAR scores by 5.5% and 5.8%, respectively.

Robustness. Applying PAFL to the baselines achieved equal or better performance for the majority of the versions of the projects. Table 5 presents the ratio of versions in each project that applying PAFL achieved equal or better performance in terms of FR. For example, in *cppcheck*, applying PAFL to Ochiai shows equal or better FR for 93% of the versions. In total, applying PAFL showed equal or better performance for 93% on average.

[2] In DL-based approaches, memory requirements increase with the size of Test Loc. This is because these approaches need to process matrices whose sizes grow significantly with the number of covered lines.

Table 5. Ratio of versions in each project that applying PAFL achieves equal or better FR than the baselines.

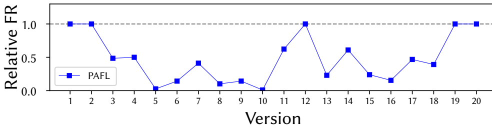
	Ochiai	DStar	Barinel	MLP-FL	CNN-FL	RNN-FL	LLMAO
cpp-peglib	100%	100%	100%	94%	92%	94%	100%
cppcheck	93%	87%	90%	69%	N/A	N/A	77%
exiv2	100%	100%	100%	87%	N/A	N/A	90%
libchewing	100%	100%	100%	90%	72%	95%	75%
libxml2	100%	100%	100%	83%	N/A	N/A	86%
proj	100%	96%	100%	69%	N/A	N/A	71%
openssl	96%	92%	96%	N/A	N/A	N/A	65%
yaml-cpp	100%	100%	100%	90%	N/A	N/A	80%
thefuck	100%	100%	97%	99%	97%	97%	100%
fastapi	100%	100%	100%	96%	93%	95%	100%
spacy	100%	100%	100%	89%	89%	89%	86%
YouTube-dl	100%	94%	100%	93%	N/A	N/A	94%
total	99%	96%	98%	86%	92%	95%	85%

Fig. 7 shows the detailed results of PAFL for the 12 projects when Ochiai was used as a baseline fault localizer. In the plots, the blue lines describe how the relative first rank (FR) of PAFL compared to Ochiai changed along with the version updates. The x-axis represents the versions (higher indicates a later version), and the y-axis represents the relative FR score (lower indicates a greater improvement in FR). If the FR scores of Ochiai and PAFL are 10 and 5, respectively, the relative FR is 0.5 (i.e., applying PAFL improved the FR score). In *exiv2* project in Fig 7a, for example, PAFL showed the same FR score with the baseline (i.e., relative FR is 1.0) in the first version as there is no past buggy version to mine fault patterns (i.e., $\Pi = \emptyset$). A few versions later (i.e., version = 3), PAFL started to show better performance than the baseline as it mined and used fault patterns. Overall, PAFL achieved strictly better performance for 15 out of 20 versions (75.0%) in the *exiv2* project.

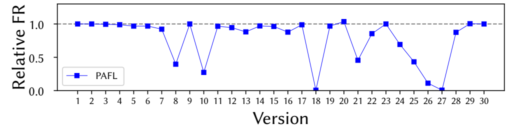
We also checked that the synthesized crosswords converged after a certain number of versions. For example, when localizing the fault in *cppcheck*#21, three crosswords synthesized from *cppcheck*#16, *cppcheck*#18, and *cppcheck*#19 are used, and the three crosswords were almost identical (they share similar suspicious token combinations).

Limitation of Our Approach. Fig. 7 also shows a limitation of PAFL. In the project *spacy* (Fig. 7l), for example, applying PAFL does not affect the performance of the baseline for the majority of the versions (85.7%, respectively). We found this was because the buggy versions have different fault patterns (i.e., the fault patterns were not redundant); thus, PAFL failed to mine and use redundant fault patterns. However, we would like to note that applying PAFL does not cause any harm to the fault localization performance, and the cost of applying PAFL is negligible, as discussed below.

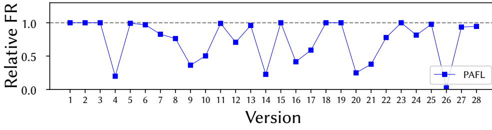
Efficiency. The overhead of PAFL was negligible. Table 6 shows the overhead cost of PAFL and the state-of-the-art fault localizer enhancing technique Aeneas [Xie et al. 2022] in minutes. The columns "PAFL" show the average overhead of PAFL (i.e., including all costs for training and updating suspiciousness scores) in minutes. The results show that the overhead of PAFL is less than 14 seconds on average. The maximum overhead was also less than one minute for the largest project, *openssl*.



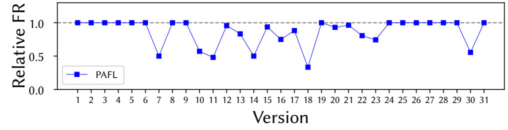
(a) Relative FR scores in *exiv2*.



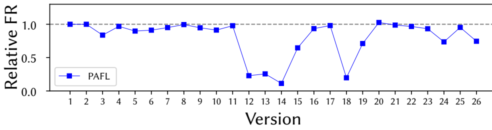
(b) Relative FR scores in *cppcheck*.



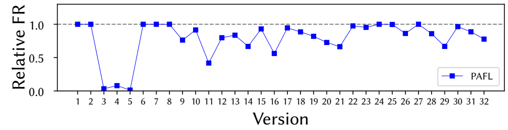
(c) Relative FR scores in *proj*.



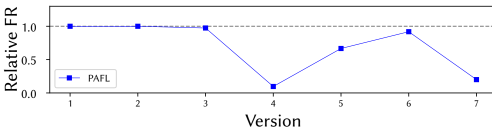
(d) Relative FR scores in *thefuck*.



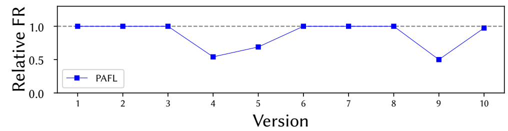
(e) Relative FR scores in *openssl*.



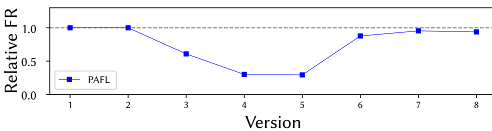
(f) Relative FR scores in *YouTube-dl*.



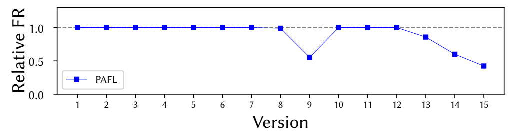
(g) Relative FR scores in *libxml2*.



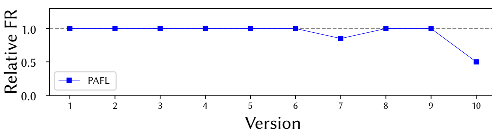
(h) Relative FR scores in *yaml-cpp*.



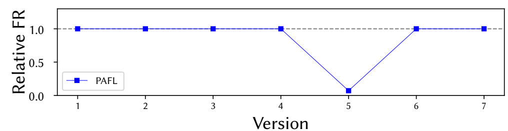
(i) Relative FR scores in *libchewing*.



(j) Relative FR scores in *fastapi*.



(k) Relative FR scores in *cpp-peglib*.



(l) Relative FR scores in *spacy*.

Fig. 7. How the relative FR scores of PAFL compared to Ochiai changed with the version updates.

4.2 RQ2: Comparison with State-of-the-Art

We compared PAFL with Aeneas [Xie et al. 2022], a state-of-the-art technique that also aims to enhance baseline fault localizers. Aeneas is a pre-processing technique that uses data augmentation to generate additional test results. To evaluate Aeneas, we used the artifact provided by the authors [Xie et al. 2022]. Aeneas has two hyper-parameters: *cp* (principle component proportion) and *ep* (largest eigenvalue proportion) [Xie et al. 2022]. Intuitively, *cp* determines the ratio of

Table 6. The average overhead of Aeneas and PAFL for each project in minutes. "N/A" indicates that the cost is unavailable because of internal errors or it exceeded the time budget 24 hours.

	Aeneas	PAFL		Aeneas	PAFL
cpp-peglib	0.50	0.01	*openssl*	N/A	0.22
cppcheck	248.26	0.03	*yaml-cpp*	3.02	0.01
exiv2	N/A	0.01	*thefuck*	8.62	0.01
libchewing	2.14	0.01	*fastapi*	1.25	0.01
libxml2	750.55	0.05	*spacy*	14.7	0.03
proj	N/A	0.03	*YouTube-dl*	10.91	0.02

Table 7. Performance of Aeneas, PAFL, and together (Aeneas+PAFL) for the SBFL techniques.

	Scenario	MFR	MAR	Top-1	Top-5	Top-10
Ochiai	original	749.6	931.8	7	21	30
	Aeneas	1926.3	2477.0	7	24	33
	PAFL	**702.1**	**879.6**	10	**30**	**39**
	together	1819.2	2163.1	**11**	26	33
DStar	original	1005.6	1230.5	8	24	33
	Aeneas	3112.6	3730.9	1	6	7
	PAFL	**884.8**	**1092.1**	**11**	**34**	**40**
	together	2872.8	3329.5	2	8	10
Barinel	original	718.7	899.4	5	14	24
	Aeneas	697.2	1206.5	4	19	29
	PAFL	661.7	**831.6**	**11**	28	**43**
	together	**617.5**	940.1	9	**31**	42

statements to be removed, and *ep* affects which statements are removed. For example, if *cp* is 0.9, 10% of the statements are removed using the *ep* value. To choose *cp* and *ep*, we ran Aeneas on our benchmarks with various configurations, $cp \in \{0.95, 0.9, 0.85, 0.8\}$ and $ep \in \{0.8, 0.75, 0.7\}$, and selected the parameters that achieved the best performance in terms of MFR. The evaluation of Aeneas was conducted using an NVIDIA RTX A6000 GPU with 48 GB of memory.

Performance Comparison. Table 7 shows the performance of Aeneas and PAFL for the three SBFL techniques. The rows "Aeneas" and "PAFL" present the performance when Aeneas and PAFL were applied, respectively. As Aeneas is orthogonal to PAFL, both pre-processing and post-processing can be applied together. The rows "Together" show the performance of the combination.

Compared to Aeneas, PAFL was far more effective and robust. Except for the baseline Barinel with the metric MFR, PAFL showed far better performance than Aeneas. In DStar, PAFL improved the baseline, while Aeneas degraded the performance in all the metrics. We checked that this was because Aeneas removed ground-truth fault statements from the candidates using the two hyper-parameters *cp* and *ep*. The baseline localizer could not give suspiciousness scores to the fault statements; it assigned the worst ranking to them. We would like to note that the table reports the performance of the best configuration of the two hyper-parameters in the 12 ($cp \in \{0.95, 0.9, 0.85, 0.8\}$ and $ep \in \{0.8, 0.75, 0.7\}$) candidate configurations.

Compared to using Aeneas and PAFL together (i.e., "together" in Table 7), PAFL alone achieved overall better performance. The combination showed better performance in only three cases: MFR of Barinel, Top-5 of Barinel, and Top-1 of Ochiai. The combination even degraded the

Table 8. Performance comparison of PAFL when the extended pattern-description language *ExtendedCW* introduced in Section 3.2 is used. In the language, the number of predecessor, successor, parent, and child nodes can be configured; we consider seven configurations $(1,1,1,1)$, $(0,0,0,0)$, $(2,1,1,1)$, $(1,2,1,1)$, $(1,1,2,1)$, $(1,1,1,2)$, $(2,2,2,2)$, where (p,q,r,s) denotes the number of predecessor, successor, parent, and child nodes, respectively. A column (p,q,r,s) presents the performance of PAFL when the configuration is used.

Metric	(1,1,1,1)	(0,0,0,0)	(2,1,1,1)	(1,2,1,1)	(1,1,2,1)	(1,1,1,2)	(2,2,2,2)
MFR	2441.7	2463.0	2444.0	2436.7	**2433.9**	2441.6	2437.7
MAR	2895.7	2933.3	2888.1	2883.8	2883.1	2895.7	**2863.0**
Top-1	**14**	11	**14**	13	13	12	**14**
Top-5	**36**	32	**36**	33	34	**36**	34
Top-10	**49**	44	**49**	**49**	**49**	**49**	47

baselines in eight cases. This shows that blindly combining the techniques does not always improve performance. To be effective, the two techniques need to be carefully combined. We leave this as future work.

Cost Comparison. PAFL was also far more efficient than Aeneas. Table 6 compares the overhead of Aeneas and PAFL in minutes. In the table, the columns "Aeneas" and "PAFL" present the average overhead (i.e., total cost − baseline localization cost) of the two techniques in minutes. The time budget for the overhead was 24 hours. "N/A" indicates that Aeneas failed its procedure due to its internal errors (*exiv2*) or it exceeded the time budget (*proj* and *openssl*). As the table shows, PAFL was significantly faster than Aeneas. In the two projects *proj* and *openssl*, for example, Aeneas failed to finish its task within the time budget, while PAFL finished its task within 2 and 13 seconds, respectively, on average.

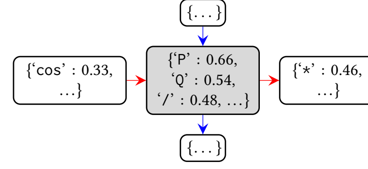
4.3 RQ3: Effect of Language Extension

Now, we experimentally evaluate how the extension of our fault pattern-description language affects the performance of PAFL. As an extended language, we consider *ExtendedCW* introduced in Section 3.2. Unlike the original language, a center node in *ExtendedCW* can have multiple predecessor, successor, parent, and child nodes. Let p, q, r, s be the number of predecessor, successor, ancestor (parent), and descendant (child) nodes, respectively. Then, an infinite number of possible configurations exist. Among them, we consider seven configurations: $(1,1,1,1)$, $(0,0,0,0)$, $(2,1,1,1)$, $(1,2,1,1)$, $(1,1,2,1)$, $(1,1,1,2)$, and $(2,2,2,2)$. For example, the configuration $(1,1,1,1)$ is the same as our original language *Crossword* where a center node has one predecessor, successor, parent, and child node. In the configuration of $(0,0,0,0)$, a center node does not have any adjacent nodes; a crossword updates the suspiciousness of statements without considering the adjacent statements. In the last configuration $(2,2,2,2)$, on the other hand, a center node has a successor that has a successor, a predecessor that has a predecessor, a parent that has a parent, and a child that has a child node. As a baseline fault localization technique, we used Ochiai [Abreu et al. 2006].

Table 8 compares the overall performance of the seven configurations for the given metrics. In the table, a column (p,q,r,s) presents the performance of PAFL when the configuration is used. The evaluation results show that using more expressive languages does not necessarily improve the performance of PAFL. The seven configurations show competitive performance across the metrics. For example, the configuration $(2,2,2,2)$, the most expressive fault pattern-description language, does not always show the best performance. In terms of the Top-1, Top-5, and Top-10 in total, the configurations $(1,1,1,1)$ and $(1,2,1,1)$ show overall the best performance.

```
54 ...
55 cosl = cos(lp.lam);
56 - k = P->k0 * Q->R2 / (...);
   + const double denom = ...;
   + if( denom == 0.0 ) {...}
   + k = P->k0 * Q->R2 / denom;
57 xy.x = k * cosc * sin(lp.lam);
58 ...
```

(a) Fault and fix in *proj*#4.



(b) Synthesized crossword from *proj*#4.

Fig. 8. Fault appeared in *proj*#4 and the used crossword for the fault localization.

Interestingly, the configuration $(0, 0, 0, 0)$, which considers only the embedding (i.e., the set of tokens) of the target statement, shows a competitive performance. In the Top-1, for example, the configuration $(0, 0, 0, 0)$ achieved only 3 less than the best configurations. This shows, that the statement embedding method is important in fault localization, and our embedding method was effective. In this work, we adopt a simple and straightforward approach, representing each statement using the set of tokens it contains. However, employing a more advanced embedding method could enhance PAFL's performance. Exploring or developing better embedding techniques is an interesting future work.

4.4 RQ4: Learned Insight

The synthesized crosswords offer developers valuable insights into the fault patterns within the project. Note that the crosswords are interpretable, describing suspicious token combinations that guide developers on where to search for specific bugs. For example, Fig. 8b shows the crossword synthesized from the *proj*#4 (causing division by 0) in Fig. 8a. From the crossword, developers can learn an insight that the combination of the tokens 'cos' and 'P' in two sequencially connected statements is suspicious and should be inspected when division by 0 occurs. For example, the developers can use the learned insight to localize the fault statement occurring division by 0 in the following recent buggy version of the *proj* project[3]:

```
120  ...
121  cosphi = cos(Q->phi2);
122 - Q->n = (m1 - pj_msfn(sinphi, cosphi, P->es)) / (pj_mlfn(...) - ml1);
    + const double ml2 = pj_mlfn(Q->phi2, sinphi, cosphi, Q->en);
    + if (ml1 == ml2) {...}
    + Q->n = (m1 - pj_msfn(sinphi, cosphi, P->es)) / (ml2 - ml1);
123  if (Q->n == 0) {
124  ...
```

In the above buggy version, the fault statement includes the token 'P' (also includes 'Q' and '/' mapped to a high suspicious score in Fig. 8b) and the previous statement includes the token 'cos'.

5 Related Work

Fault localization has been widely studied in the literature [Kang et al. 2024; Li et al. 2022a,b; Lou et al. 2020; Rafi et al. 2024; Shao and Yu 2023; Soremekun et al. 2023; Wen et al. 2016; Yang et al. 2024; Zeng et al. 2022]. We discuss those works that are closely related to ours.

[3]https://github.com/OSGeo/PROJ/commit/586ef0f5f1099fa7aaaa353334e15871ab985127#diff-51299374e446ed8116e375ba3c05709f5ed917dc5450f4baa5fa29af52008e8fR114

Machine Learning-Based Fault Localization. Machine learning-based techniques have been widely used for fault localization [Jiang et al. 2023; Li et al. 2019; Sohn and Yoo 2017; Widyasari et al. 2022; WONG and QI 2009]. For example, FLUCCS [Sohn and Yoo 2017] uses a genetic algorithm to learn a model from the code features and the spectrum data. ABLFL [Pan et al. 2020], a deep learning-based fault localization technique, uses both static features (e.g., statistical information of source code) and dynamic features (e.g., program execution information) to learn a fault localization model. XAI4FL [Widyasari et al. 2022] learns the importance of failing test cases for enhancing SBFL techniques. HSFL [Wen et al. 2021] is also closely related to ours. HSFL improves SBFL techniques by identifying and using bug-inducing commits from past versions.

Theoretically, existing machine learning-based approaches can learn a project-specific model if trained exclusively on past versions of the target project. However, the learning-based approaches require a large amount of training data (e.g., 190 buggy versions [Sohn and Yoo 2017]) to develop a model that generalizes well, which significantly limits their applicability to project-aware fault localization. In our evaluation, we used the projects that have at most 30 buggy versions, which is significantly less than the required amount of training data for the learning-based approaches. Also, PAFL is orthogonal to these approaches, as it can be combined with them to further improve their performance.

Other Fault Localization Techniques. Our approach can also be combined with other flavors of fault localization techniques. For example, PAFL can be applied to other spectrum-based fault localization techniques such as TARANTULA [Jones et al. 2002], ZOLTAR [Janssen et al. 2009], and JACCARD [Abreu et al. 2007]. The techniques use different formulas to compute the suspiciousness of the program statements. Mutation-based fault localization techniques [Hong et al. 2015; Li and Zhang 2017; Moon et al. 2014; Papadakis and Le Traon 2015] can also be used in our approach. For example, MUSEUM [Hong et al. 2015], a multilingual mutation-based fault localization technique, can be used as a baseline localizer. However, information retrieval-based approaches [Miryeganeh et al. 2021; Moreno et al. 2014; Saha et al. 2013; Shao and Yu 2023; Wen et al. 2016] may not be suitable for our approach, as they typically use coarser-grained levels (e.g., file-level) rather than statement-level fault localization.

6 Conclusion

In this paper, we investigated a project-aware fault localization approach for enhancing existing fault localizers by enabling them to leverage project-specific fault patterns. Our technique, PAFL, is based on two novel ideas. First, we designed a simple domain-specific language to describe various fault patterns. Second, we developed a synthesis algorithm for mining project-specific fault patterns in terms of crosswords from past versions of the target project. The evaluation results show that our approach can effectively, robustly, and efficiently boost a variety of baseline fault localizers including spectrum-based and deep learning-based approaches.

Future Work. In this paper, we implemented and evaluated PAFL primarily for C/C++ programs, as this work originated from a collaboration with a company whose codebases are written in C++. However, our approach is fundamentally language-independent and can be readily applied to other languages, such as Java.

Data-Availability Statement

The artifact of PAFL is available in Zenodo [Kim 2025] and Github[4]. The artifact includes the implementation of PAFL, the installation guide for the benchmarks, and the evaluation scripts.

[4]https://github.com/kupl/PAFL

Acknowledgments

This work was supported by Samsung Electronics Co., Ltd (Project Num: IO230117-04672-01). This work was also supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2024-00333885, RS-2021-NR060080), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair, 40%, No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains, 5%), and ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2025-RS-2020-II201819, 5%).

References

Rui Abreu, Peter Zoeteweij, and Arjan J.c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. 39–46. doi:10.1109/PRDC.2006.18

Rui Abreu, Peter Zoeteweij, and Arjan J.C. 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)*. 89–98. doi:10.1109/TAIC.PART.2007.13

Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 88–99. doi:10.1109/ASE.2009.25

Gabin An, Minhyuk Kwon, Kyunghwa Choi, Jooyong Yi, and Shin Yoo. 2023. BUGSC++: A Highly Usable Real World Defect Benchmark for C/C++. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*. 2034–2037. doi:10.1109/ASE56229.2023.00208

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* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 177–188. doi:10.1145/2931037.2931049

Junming Cao, Shouliang Yang, Wenhui Jiang, Hushuang Zeng, Beijun Shen, and Hao Zhong. 2020. BugPecker: Locating Faulty Methods with Deep Learning on Revision Graphs. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1214–1218.

Wen Chen, Wei Zheng, Desheng Hu, and Jing Wang. 2016. Fault Localization Analysis Based on Deep Neural Network. *Mathematical Problems in Engineering* 2016 (2016), 1820454. doi:10.1155/2016/1820454

Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. doi:10.1109/TSE.2017.2755013

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)*. 464–475. doi:10.1109/ASE.2015.14

Tom Janssen, Rui Abreu, and Arjan J.C. van Gemund. 2009. Zoltar: a spectrum-based fault localization tool. In *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime* (Amsterdam, The Netherlands) *(SINTER '09)*. Association for Computing Machinery, New York, NY, USA, 23–30. doi:10.1145/1596495.1596502

Jiajun Jiang, Yumeng Wang, Junjie Chen, Delin Lv, and Mengjiao Liu. 2023. Variable-based Fault Localization via Enhanced Decision Tree. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 41 (dec 2023), 32 pages. doi:10.1145/3624741

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 2002*. 467–477. doi:10.1145/581396.581397

Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proc. ACM Softw. Eng.* 1, FSE, Article 64 (jul 2024), 23 pages. doi:10.1145/3660771

Donguk Kim. 2025. PAFL: Enhancing Fault Localizers by Leveraging Project-Specific Fault Patterns (Artifact). doi:10.5281/zenodo.14920999

Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 165–176. doi:10.1145/2931037.2931051

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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180. doi:10.1145/3293882.3330574

Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 92 (oct 2017), 30 pages. doi:10.1145/3133916

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* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 661–673. doi:10.1109/ICSE43902.2021.00067

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022a. 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 659–671. doi:10.1145/3540250.3549137

Zeyan Li, Nengwen Zhao, Mingjie Li, Xianglin Lu, Lixin Wang, Dongdong Chang, Xiaohui Nie, Li Cao, Wenchi Zhang, Kaixin Sui, Yanhua Wang, Xu Du, Guoqiang Duan, and Dan Pei. 2022b. Actionable and interpretable fault localization for recurring failures in online service systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 996–1008. doi:10.1145/3540250.3549092

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. doi:10.1145/3293882.3330577

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* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 75–87. doi:10.1145/3395363.3397351

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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 664–676. doi:10.1145/3468264.3468580

Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving Fault Localization and Program Repair with Deep Semantic Features and Transferred Knowledge. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1169–1180. doi:10.1145/3510003.3510147

Nima Miryeganeh, Sepehr Hashtroudi, and Hadi Hemmati. 2021. GloBug: Using global data in Fault Localization. *Journal of Systems and Software* 177 (2021), 110961. doi:10.1016/j.jss.2021.110961

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*. 153–162. doi:10.1109/ICST.2014.28

Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 151–160. doi:10.1109/ICSME.2014.37

Yoann Padioleau. 2009. Parsing C/C++ Code without Pre-processing. In *Compiler Construction*, Oege de Moor and Michael I. Schwartzbach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 109–125.

Yuqing Pan, Xi Xiao, Guangwu Hu, Bin Zhang, Qing Li, and Haitao Zheng. 2020. ALBFL: A Novel Neural Ranking Model for Software Fault Localization via Combining Static and Dynamic Features. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 785–792. doi:10.1109/TrustCom50675.2020.00107

Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test. Verif. Reliab.* 25, 5–7 (aug 2015), 605–628. doi:10.1002/stvr.1509

Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 609–620. doi:10.1109/ICSE.2017.62

Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. 2024. Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation. *Proc. ACM Softw. Eng.* 1, FSE, Article 86 (jul 2024), 23 pages. doi:10.1145/3660793

Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 345–355. doi:10.1109/ASE.2013.6693093

S. Shao and T. Yu. 2023. Information Retrieval-Based Fault Localization for Concurrent Programs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1467–1479. doi:10.1109/ASE56229.2023.00122

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* (Santa Barbara, CA, USA) *(ISSTA 2017)*.

Association for Computing Machinery, New York, NY, USA, 273–283. doi:10.1145/3092703.3092717

Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Mike Papadakis. 2023. Evaluating the Impact of Experimental Assumptions in Automated Fault Localization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 159–171. doi:10.1109/ICSE48619.2023.00025

Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2348–2368. doi:10.1109/TSE.2019.2948158

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

Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus A. Haryono, Yuan Tian, Hafil Noer Zachiary, and David Lo. 2022. XAI4FL: Enhancing Spectrum-Based Fault Localization with Explainable Artificial Intelligence. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 499–510. doi:10.1145/3524610.3527902

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. doi:10.1145/3368089.3417943

W. Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. 2012. Software Fault Localization Using DStar (D*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*. 21–30. doi:10.1109/SERE.2012.12

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. doi:10.1142/S021819400900426X arXiv:https://doi.org/10.1142/S021819400900426X

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. doi:10.1145/3540250.3549101

Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. 2022. A Universal Data Augmentation Approach for Fault Localization. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 48–60. doi:10.1145/3510003.3510136

Xiaoyuan Xie, Fei Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. 2013. Provably optimal and human-competitive results in SBSE for spectrum based fault localisation. In *Search Based Software Engineering - 5th International Symposium, SSBSE 2013, Proceedings (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics))*. 224–238. doi:10.1007/978-3-642-39742-4_17 5th International Symposium on Search-Based Software Engineering, SSBSE 2013 ; Conference date: 24-08-2013 Through 26-08-2013.

Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. doi:10.1145/3597503.3623342

Bo Yang, Yuze He, Huai Liu, Yixin Chen, and Zhi Jin. 2020. A Lightweight Fault Localization Approach based on XGBoost. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. 168–179. doi:10.1109/QRS51102.2020.00033

He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 13 pages. doi:10.1145/3597503.3623337

Yanbing Yu, James Jones, and Mary Jean Harrold. 2008. An empirical study of the effects of test-suite reduction on fault localization. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 201–210. doi:10.1145/1368088.1368116

Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault localization via efficient probabilistic modeling of program semantics. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 958–969. doi:10.1145/3510003.3510073

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)*. 445–455. doi:10.1109/SANER.2019.8668002

Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Xiaohong Zhang. 2021. A study of effectiveness of deep learning in locating real faults. *Information and Software Technology* 131 (2021), 106486. doi:10.1016/j.infsof.2020.106486

Received 2024-10-16; accepted 2025-02-18