From Sparse to Structured: A Diffusion-Enhanced and Feature-Aligned Framework for Coincidental Correctness Detection

Huan Xie[1], Chunyan Liu[1], Yan Lei[1]*, Zhenyu Wu[1], Jinping Wang[1]
[1]School of Big Data and Software Engineering, Chongqing University, Chongqing, China
Email: {huanxie, chunyanliu, yanlei}@cqu.edu.cn, {zhenyu_wu, jpwang}@stu.cqu.edu.cn

***Abstract*— Coincidental correctness (CC) refers to test cases that execute faulty code but still produce excepted outputs. This phenomenon introduces noise into the data of software testing-related tasks. As demonstrated in the literature, CC has negative impact on test suite reduction, test case prioritization, fault localization, and automated program repair. Thus, it is essential to detect and mitigate the impact of CC. Although CC is commonly observed across a large number of programs, CC test cases are typically sparse within each program's test suite. In other words, CC test cases generally make up merely a small portion of the passing test cases. The proportions vary from 3.27% to 31.74% within Defects4J V1.4. This results in a highly imbalanced distribution of CC versus non-CC test cases, posing challenges for accurate detection.**

To address this issue, we propose a Diffusion-Enhanced and Feature-Aligned Framework for Coincidental Correctness detection, named *DEFACC*, to obtain more structured representations of test cases. Specifically, *DEFACC* first introduces a diffusion-based generation module. This module generates new CC samples from original samples to alleviate class imbalance issue and enhance the diversity of CC samples. However, generated feature samples may deviate from the distribution of real CC samples. Such shifts can hurt model reliability and generalization. To resolve this, *DEFACC* integrates a feature alignment module that is founded on the *Maximum Mean Discrepancy (MMD)* loss. This module enforces distributional consistency between generated and original CC samples during training. Together, these components ensure that the augmented samples are from sparse to structured, which is not only quantitatively balanced but also semantically faithful. Experimental results show that the *DEFACC* significantly improves the performance of existing CC detection methods and provides a stronger representation foundation for accurate fault localization.

***Index Terms*—Coincidental correctness detection, data imbalance, fault localization, diffusion model, feature alignment**

I. Introduction

Coincidental correctness (CC) [1]–[4] refers to a test that executes faulty code but still produces the expected output. Generally, the test cases along with their corresponding results serve as the source data for many software testing-related tasks. The presence of CC introduces noise into such source data. Source data with noise could have a negative impact on these software testing-related tasks. Previous research has demonstrated that CC negatively influences a range of software testing-related tasks, including test suite reduction [4], test case prioritization [4], fault localization [5], and automated program repair [6].

To mitigate the impact of CC, prior studies [2], [5], [7]–[11] have primarily adopted two types of methods to identify CC test cases in datasets: heuristic-based algorithms [2], [7] and machine learning-based algorithms [5], [8]–[11]. Although these methods have achieved improvements on some intended aspects of CC detection, they often overlook the severe class imbalance present in the dataset. Specifically, in the real world, CC test cases comprise only a small portion of all passing tests.

For clarity and consistency, we first define three key terms. Following Xie et al. [12], we categorize test cases into three groups: (1) **genuine passing tests**, which do not execute faulty entities and pass; (2) **CC tests**, which execute faults but pass; and (3) **failing tests**, which execute faults and fail. To better understand the distribution of CC within the test suite, we conduct a statistical analysis on the Defects4J V1.4 benchmark. Table I reports the detailed results, where 'GP' refers to genuine passing tests and '$\frac{|CC|}{|CC+GP|}$' denotes the proportion of CC tests among all passing tests. The results show that CC tests account for as little as **3.27%** and at most only **31.74%** of passing tests, highlighting a severe class imbalance. This imbalance may significantly impair the CC detection model's ability to learn CC-specific features, leading to CC misclassification.

TABLE I: The distribution of failing tests, CC tests, and genuine passing tests in Defects4J

Program	Tests	Failing Tests	CC	GP	$\frac{\lvert CC\rvert}{\lvert CC+GP\rvert}$
Chart	3,565	92	484	2,989	13.94%
Closure	241,642	4,457	47,441	189,744	20.00%
Lang	4,589	124	737	3,728	16.55%
Math	12,089	176	1,830	10,083	15.37%
Mockito	20,200	120	6,371	13,709	31.74%
Time	50,933	76	1,661	49,196	3.27%
Total	**333,018**	**5,045**	**58,524**	**269,449**	**17.84%**

Despite increasing interest in CC detection, prior work [2], [9]–[11] rarely addresses this imbalance directly. Most approaches typically use such imbalanced datasets directly as input to CC detection models, without explicitly addressing the imbalance itself. Moreover, generating new CC tests at the test case level is challenging and often fails to preserve

*Yan Lei is corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00041

semantic fidelity.

To tackle these issues, we propose ***DEFACC***, a generative framework designed to enrich the representation of CC tests at the feature level. Specifically, ***DEFACC*** consists of two core modules and three major stages. First, in the data preparation stage, ***DEFACC*** encodes each test case into a fixed-dimensional feature vector by concatenating three types of features: expert, coverage, and semantic features. Next, in the feature augmentation stage, a diffusion module generates CC feature vectors within the representation space, capturing the semantics and structural patterns of real CC tests. To address potential distributional shifts between real and generated features, ***DEFACC*** introduces a representation alignment stage. Here, a *Maximum Mean Discrepancy (MMD)* loss is applied to align the distributions, ensuring that generated CC features maintain semantic consistency with original CC instances. In summary, ***DEFACC*** not only resolves the imbalance of CC features but also ensures their semantic alignment with real CC instances, producing balanced and semantically meaningful training data. Beyond these benefits, it is crucial to emphasize that ***DEFACC*** is a **feature-level augmentation framework**, not a standalone CC detector. It can be applied to any feature-based CC detector. For each CC detection method, ***DEFACC*** retain its original feature extraction process and train the diffusion model in the same feature space. Then, it generates pseudo-CC samples compatible with the detector's native features and incorporates them into the training data to improve learning. This flexible design ensures that ***DEFACC*** can effectively enhance diverse CC detectors without requiring any changes to their architectures.

We apply our framework to four existing four state-of-the-art CC detection methods. On the Defects4J dataset, experimental results show that our framework can improve the CC detection effectiveness compared to the original methods. Since many CC-related studies [5], [6], [9], [11]–[14] adopt fault localization (FL) as the downstream task, we next validate the effectiveness of our framework on FL. The results also show it is helpful to improve the effectiveness of FL.

To sum up, this paper makes the following contributions:

- We are the first to address CC data imbalance using a feature-level generative augmentation method.
- We propose ***DEFACC***, a diffusion-enhanced and feature-aligned framework for CC detection. The framework integrates a bit diffusion model to generate CC features and a feature alignment module to reduce distributional divergence between generated and real CC features. By aligning these features in the latent space, generated CC features maintain semantic consistency and improve the accuracy of CC detection.
- We conduct a comprehensive evaluation of ***DEFACC*** on the Defects4J benchmark. Experimental results indicate that ***DEFACC*** has the capacity to enhance the effectiveness of four CC detection and six FL baselines across diverse programs.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III presents the details of our proposed approach. Section IV and Section V report the experimental setup and evaluation results. Section VI discusses the limitations and further implications. Finally, Section VII concludes the paper.

II. RELATED WORK

A. Diffusion Model

Diffusion models [15]–[18] have recently achieved remarkable success in a wide range of generative tasks, such as image synthesis [15], [17], [19], [20], video generation [21]–[26], and natural language processing [27]–[30]. The core idea of diffusion models is to gradually corrupt data by adding noise and then learn to reverse this process to recover high-quality samples. Compared to *Generative Adversarial Networks (GANs)* [19] and *Variational Autoencoders (VAEs)* [31], diffusion models offer better training stability, broader mode coverage, and higher output quality. These advantages have led to increasing interest in their applications to structured data [31], [32] and low-resource tasks [33]–[35].

In the context of CC detection, a key challenge lies in modeling extremely imbalanced class distributions in structured feature spaces. Standard Gaussian-based diffusion models [31], primarily designed for continuous data, often fail to capture the semantics of discrete structures such as code coverage vectors or feature embeddings. This mismatch may lead to issues such as semantic drift or mode collapse, especially when generating rare-class samples. Moreover, under extreme class imbalance, it is difficult for these models to generate samples that are both diverse and class-consistent.

Chen et al. [18] proposed the *bit diffusion model*, which is specifically tailored for discrete data generation. This approach encodes discrete features into binary form and then maps them to a real-valued space using a simple transformation ($0 \rightarrow$ -1.0, $1 \rightarrow$ 1.0), enabling compatibility with continuous diffusion frameworks. Bit-level modeling captures high-order semantic dependencies and supports class-conditional generation, resulting in greater fidelity and consistency for minority-class synthesis.

Inspired by this approach, the ***DEFACC*** framework adopts the bit diffusion model to generate high-quality CC feature samples. Its goal is to improve class balance and facilitate downstream feature learning. The feature space in ***DEFACC*** combines three types of data: expert features (e.g., suspiciousness scores), coverage features (e.g., execution paths), and semantic features (e.g., code embeddings). These features often include discrete or sparse semantics, which demands strong modeling and control capabilities, and a bit diffusion model fits this task.

As shown in Fig. 1, each test case is first encoded as three types of binary feature vectors (i.e., semantic, expert, and coverage). In the training phase, these binary values are transformed into real-valued vectors using the rule: 0 becomes -1.0 and 1 becomes +1.0. The resulting real-valued vectors are then used to train the bit diffusion model.

In the generation phase, the trained model takes a class-conditional vector as input and produces a generated real-

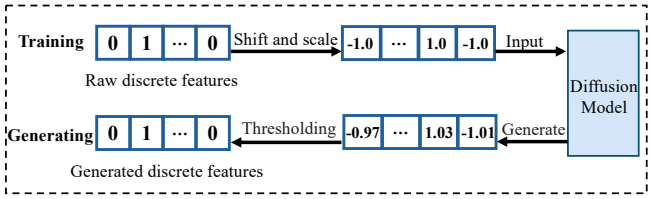


Fig. 1: Architecture of Bit Diffusion-Based Conditional Generation for CC Feature Synthesis

valued vector (e.g., -1.01, +1.03, -0.97). A thresholding operation is then applied: values ≥ 0 are mapped to 1, and values 0 are mapped to 0. This produces a generated CC feature vector in binary form, structurally aligned and semantically consistent with real CC features.

B. Feature Alignment

Feature alignment has emerged as a critical strategy in machine learning, especially in the contexts of transfer learning [36], [37] and imbalanced learning [38]–[40], where discrepancies in data distributions can lead to significant generalization degradation. The key objective of feature alignment is to reduce the distribution gap between different domains or categories, thereby enhancing the model's robustness and discriminative ability when encountering unseen or rare samples.

Existing alignment methods broadly fall into two categories. One class leverages adversarial learning, such as *Domain-Adversarial Neural Networks (DANN)* [41], which employ a domain discriminator to guide the feature extractor to generate domain-invariant representations. The other class utilizes statistical distance measures like MMD [42] or Kernel Alignment [43]. These statistical approaches are often more stable in training and require no adversarial loss, making them attractive for practical applications.

In the context of data generation and augmentation, feature alignment has also been widely adopted to enhance the semantic consistency and distributional fidelity of generated samples [19], [44]. Particularly when generative models (e.g., diffusion models) are used to generate rare-class data, ignoring the discrepancy between generated and real distributions can cause overfitting, semantic drift, or class confusion during training. To address this issue, recent studies [45]–[47] have explored the integration of feature alignment into software engineering tasks. For example, Xu et al. [45] proposed a model that extracts and aligns representative features for adaptive object detection. Xie et al. [46] presented *FEADet* with an Adaptive AlignConv module to achieve feature alignment for oriented object detection. Li et al. [47] proposed *MMD GAN*, where MMD measures the difference between the generated and real distributions, guiding the generator training to improve the quality and diversity of generated samples in image generation tasks.

In our work, we consider that CC features generated by a bit diffusion model may deviate from the real CC features distribution. We thus introduce MMD as the core metric for feature alignment. As a non-parametric, kernel-based measure, MMD offers advantages in computational efficiency and generalization. By minimizing the MMD distance between real and generated CC features, our approach ensures the generated CC features are statistically consistent with real CC features, thereby improving CC detection accuracy and reducing overfitting risk.

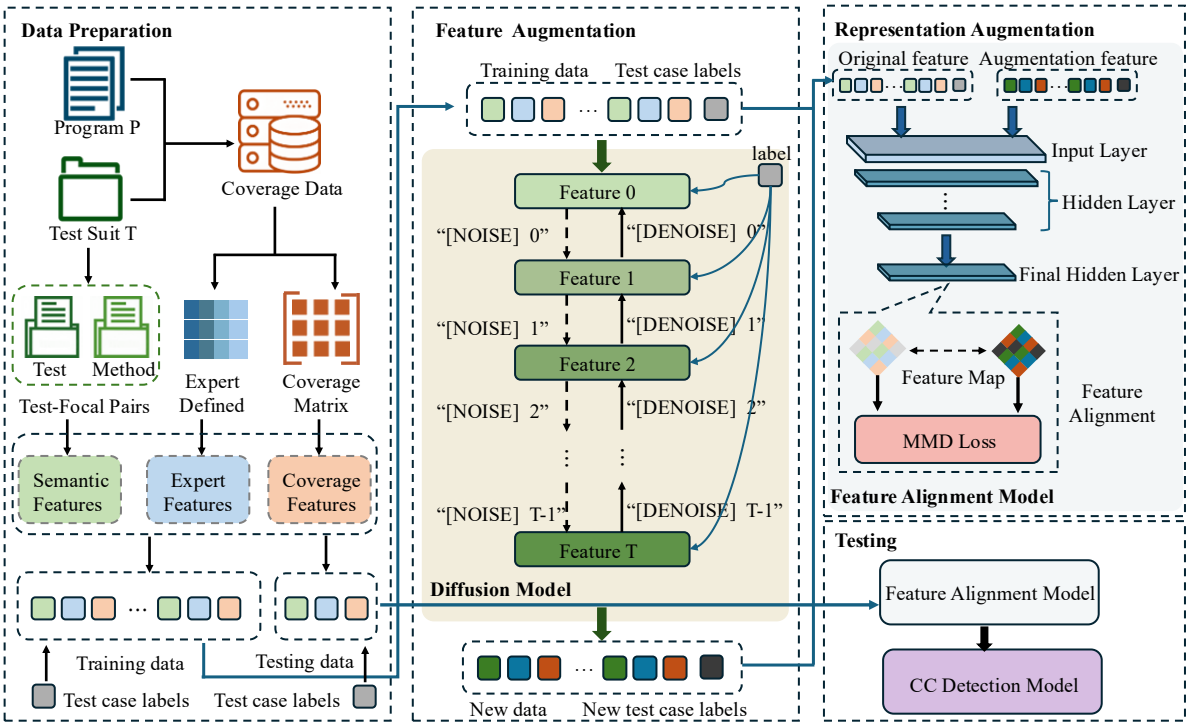
C. Coincidental Correctness

CC refers to test cases that execute faulty code but still yield passing results, which undermines the performance of model training in downstream tasks [1], [4]. Early studies employed clustering and *Support Vector Machine (SVM)*. Miao et al. [48] clustered the configuration files associated with test cases, grouping those with similar behaviors into the same cluster to detect CC. Xue et al. [49] enhanced the classification capability of machine learning models for CC detection by employing ensemble learning techniques combined with the strong generalization ability of support vector machines. Recent studies have leveraged neural network-based approaches for CC detection, including MLP-based feature integration [11], contrastive learning with MLP classifiers [50], and triplet networks using key execution features [12]. Despite progress, most existing methods overlook the severe class imbalance of CC tests in real-world datasets, which motivates our focus on feature augmentation for CC detection.

III. METHODOLOGY

A. Overview

To address the class imbalance of CC in real-world datasets, we propose ***DEFACC***, a feature-level augmentation framework for generating high-quality pseudo CC feature vectors to improve training stability and generalization. Fig. 2 shows the overview of our framework. As illustrated in Fig. 2, ***DEFACC*** consists of three main stages. In the data preparation stage (Section III-B), we extract three types of features for each test case: expert features (e.g., suspiciousness scores), coverage features via a *convolutional neural network (CNN)*, and semantic features from UniXcoder [51]. These features are concatenated into a unified representation, which serves as the input for the feature augmentation stage (Section III-C). Here, a bit diffusion model generates pseudo CC features guided by real CC features, ensuring semantic consistency and feature diversity. In the representation augmentation stage (Section III-D), a *multi-layer perceptron (MLP)* is used as an encoder to align the generated and real features in latent space. Specifically, we apply the MMD loss to minimize the distribution shift between the two. The diffusion model offers stable generation and fine-grained control, making it well-suited for rare-class generation, while the MMD loss enforces statistical alignment to reduce overfitting. These components jointly enable ***DEFACC*** to produce semantically consistent and distribution-aligned CC features, supporting robust CC detection under data-scarce conditions. In the following sections, we will detail the three stages of ***DEFACC***.



DEFACC outputs augmented feature vectors to be used by downstream CC detectors (plug-in).

Fig. 2: The architecture of ***DEFACC***

B. Data Preparation

In this section, we describe the data preparation process. For each test case, we extract three categories of features(i.e., coverage, expert, and semantic features). These features are concatenated into a unified feature vector. Together with the corresponding test label (**0** for **genuine passing tests**, **1** for **CC tests**), they serve as input to the diffusion model, leading it to generate pseudo CC features. The detail of how each feature type is extracted is in the following subsections.

1) Coverage Features: Coverage features refer to the execution information of test cases over program entities (e.g., statements, functions, or classes). They are typically represented as binary vectors or matrices. Each element denotes the execution status of a program entity, where 1 indicates execution and 0 indicates no execution. Both raw and processed coverage data (e.g., after program slicing) can be directly used as coverage features. Because coverage features reflect the behavioral similarity between failing and CC test cases, they are widely used in CC detection [5], [9], [48]. Moreover, coverage features capture structural execution patterns in the program. These patterns help the diffusion model produce pseudo CC features that reflect realistic behaviors rather than arbitrary noise. To obtain compact and expressive coverage representations, we follow prior work [52] and adopt a CNN to extract high-level features from raw coverage data. Compared with raw binary coverage vectors, the learned coverage embeddings better capture spatial locality and behavioral patterns along execution paths, improving the ability to distinguish CC tests. Specifically, the CNN architecture consists of two convolutional layers, two max pooling layers, one fully connected layer, and one output layer.

2) Expert Features: Exiting research [11], [13], [14] has proposed multiple expert features that capture how test cases behave in terms of statement coverage and their similarity to failing tests. These features provide essential signals for generating CC feature vectors. Following prior work [11], we extract three types of expert features based on ten widely-used *spectrum-based fault localization (SBFL)* formulas. Each SBFL formula contributes three types of feature values, resulting in a 30-dimensional expert feature vector. To summarize, expert features are typically categorized into three main types:

(1) Suspiciousness Score: Based on SBFL formulas [53]–[55], each statement in the program is assigned a suspiciousness score. For each passing test case t_i in the passing test set T_p, its suspiciousness score $ss(t_i)$ is calculated by averaging the suspiciousness scores of all the statements it covers. Formally, the score is defined as: $ss(t_i) = \frac{1}{|S_i|}\sum_{s_j \in S_i} sus_j$, where S_i is the set of statements covered by test t_i, and sus_j is the suspiciousness score of statement s_j, computed using an SBFL formula.

(2) Coverage Ratio: A test case that covers more highly suspicious statements is more likely to be a CC test. The coverage ratio $cr(t_i)$ quantifies the proportion of highly suspicious statements (with $sus_j > 0.5$) covered by t_i. It is defined as: $cr(t_i) = \frac{1}{|S_i|}\sum_{s_j \in S_i,\, sus_j > 0.5} x_{ij}$, where $x_{ij} \in \{0, 1\}$ indicates whether test t_i covers statement s_j. The higher the ratio, the more likely the test case is to be CC.

(3) Similarity Feature: A passing test case is more likely

to be a CC if its coverage pattern is similar to that of a failing test case. We compute the similarity feature by measuring the inverse of the weighted Euclidean distance [56] between the passing test t_i and each failing test $t_m \in T_f$. Here T_f and t_i are drawn only from the training set. The maximum similarity is used as the final score: $sf(t_i) = \min_{t_m \in T_f} \left(\sqrt{\sum_{j=1}^{N} sus_j \cdot (x_{ij} - x_{mj})^2} \right)$, where T_f is the set of failing test cases; N is the total number of program statements; $x_{ij}, x_{mj} \in \{0, 1\}$ represent whether t_i and t_m cover statement j, respectively.

These expert features capture essential behavioral traits of CC test cases, such as "highly suspicious yet passed" coverage patterns or close resemblance to failing cases. Thus, they provide meaningful behavioral signals for modeling the distribution of CC features within the generative framework.

3) Semantic Features: Semantic features aim to capture the intent-level and contextual relationships between test methods and their corresponding focal methods (i.e., the methods under test). These features are particularly useful in modeling features of *semantic correctness but behavioral failure*, which are common among CC test cases. Fig. 3 shows the process of semantic feature extraction. The process consists of two stages: (1) constructing test–focal pairs, and (2) extracting semantic features.

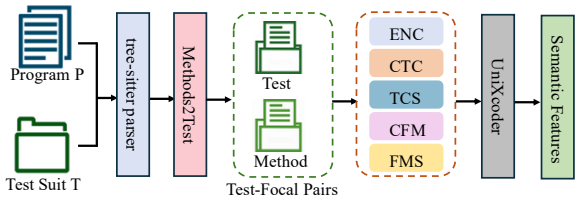


Fig. 3: Semantic Feature Extraction Process

(1) Constructing Test–Focal Pairs. Before extracting semantic features, we must identify high-quality pairs of test methods and their corresponding focal methods. We use *Tree-sitter parser* [57] and *Methods2Test* [58] to assist with this extraction. *Tree-sitter* is used to analyze both the program code and the test suite, extracting metadata about classes and methods. Based on this metadata, *Methods2Test* detects test–focal pairs. Methods2Test detects test classes using either the `@Test` annotation or filenames ending with "Tests". It then uses a set of heuristics to identify the focal method for each test case, including matching method/class names between test and code, detecting single function calls inside the test body, and filtering out weakly coupled or ambiguous mappings. Test–focal pairs that do not meet these criteria are filtered out.

(2) Extracting Semantic Embeddings. We use the pre-trained model UniXcoder to learn semantic representations for each test–focal pair. Each input sequence to UniXcoder consists of five components:

$$\text{Input} = [\text{Enc}], \text{CTC}, \text{TCS}, \text{CFM}, \text{FMS} \tag{1}$$

where: [Enc] is a special encoder token; CTC is the comment of the test method; TCS is the code body of the test method; CFM is the comment of the focal method; and FMS is the code body of the focal method. Each token is represented by a token embedding s_i and a positional embedding p_i, forming the initial embedding $h_i^0 = [s_i, p_i]$. The sequence is then passed through N Transformer layers, where each layer contains multi-head self-attention and a feed-forward network, yielding contextualized representations h_i^N. To obtain the final semantic vector, we take the first output token h_1^N and project it through three fully connected layers:

$$\text{Semantic Feature} = \text{MLP}(h_1^N) \in R^{64} \tag{2}$$

Finally, we obtain the semantic features with 64 dimensions. The resulting embeddings capture high-level semantics such as test intent, code-comment alignment, and interaction context. These semantic features offer strong guidance for our generative model to synthesize behaviorally incorrect yet semantically plausible CC samples.

4) Feature Normalization and Binarization: To ensure that the three categories of features (i.e., coverage, expert, and semantic features) contribute equally to the learning process, we apply feature normalization and feature binarization before feeding them into the bit diffusion model. Since these features are derived from heterogeneous sources and exhibit different value ranges and statistical distributions, normalization is crucial to eliminate scale differences. We adopt mean-variance normalization to scale each feature dimension to the range of $[-1, 1]$. This normalization strategy helps prevent any single feature type from dominating due to its magnitude.

Then, we convert these feature values into binary format using a thresholding function:

$$x_i = \begin{cases} 1 & \text{if } v_i > \tau \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Here, v_i denotes the original feature value, and τ is a fixed threshold (set to 0). We choose 0 as it serves as a natural midpoint in the normalized $[-1, 1]$ range, allowing a straightforward separation between high-valued and low-valued features without introducing task-specific bias.

This transformation is based on two main considerations. First, although the feature vectors are continuous, they represent discrete semantic states in a high-dimensional space. The meaning of a test case is determined by the overall configuration of the vector, rather than the precise values of individual dimensions. From this perspective, each vector can be viewed as encoding a discrete semantic identity.

Second, the bit diffusion model is specifically designed for binary inputs. Binarizing the features ensures compatibility with the model's learning dynamics and enhances its ability to generate realistic outputs. This binarization reduces noise, lowers the risk of overfitting, and improves generalization across different test scenarios.

C. Feature Augmentation

As shown in Fig. 2, we adopt the bit diffusion framework to generate new pseudo CC feature vectors from original test case features. The process consists of three key stages: *Analog Bit Encoding*, *Diffusion Training*, and *Feature Generating*.

(1) Analog Bit Encoding. Each test case is first encoded into a 158-dimensional (64 dimension coverage feature, 30 dimension expert feature, and 64 dimension semantic feature) binary vector: $x_0 \in \{0,1\}^{158}$. This vector is formed by concatenating three types of features: expert, coverage, and semantic features. To enable gradient-based training in a continuous space, we convert the binary vector into an analog representation: $x_0^{\text{analog}} \in [-1.0, +1.0]^{158}$. Specifically, each 0 in the binary vector is mapped to -1.0, and each 1 is mapped to $+1.0$. This transformation yields a real-valued vector that can be processed by the diffusion model.

(2) Diffusion Training. We follow the bit diffusion framework [18], which models the data distribution by learning to reverse a gradual noising process.

In the *forward process*, the analog input $x_0^{\text{analog}} \in R^{158}$ is perturbed with Gaussian noise over a continuous time interval $t \in [0,1]$, yielding a noisy vector x_t:

$$x_t = \gamma(t) \cdot x_0^{\text{analog}} + \sqrt{1-\gamma(t)^2} \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{4}$$

Here, $\gamma(t)$ is a monotonic decreasing noise schedule (e.g., cosine), controlling the contribution of the clean signal, and ϵ is sampled from the standard normal distribution. The process gradually transforms the original input into pure noise as $t \to 1$.

To reverse this process, we train a conditional denoising model $f_\theta(x_t, t, y)$, where x_t is the noisy input at time t, $y \in \{0,1\}$ is the binary class label (1 for CC, 0 for non-CC), and θ represents model parameters. The model learns to predict the original analog input: $\hat{x}_0 = f_\theta(x_t, t, y)$.

The model is trained by minimizing the *mean squared error (MSE)* between the predicted and original analog vectors over random time steps: $\mathcal{L}_{\text{diff}} = E_{x_0,t,\epsilon}\left[\|f_\theta(x_t,t,y) - x_0\|_2^2\right]$. This objective encourages the model to recover x_0 from any noisy intermediate state x_t, conditioned on the class label y.

(3) Feature Generating. After training, we use the learned model to generate new pseudo CC feature vectors. Starting from a Gaussian noise vector: $x_T \sim \mathcal{N}(0, I)$. We iteratively apply the denoising function in reverse, stepping from $t = T$ to $t = 0$, conditioned on $y = 1$ (i.e., CC class). This produces a reconstructed analog vector: $\hat{x}_0 \in [-1.0, +1.0]^{158}$.

Finally, we apply a sign function to discretize the analog output into a binary-like vector: $\hat{x}^{\text{CC}} = \text{sign}(\hat{x}_0) \in \{0,1\}^{158}$. where each value greater than 0 is mapped to 1, and less than 0 to 0. These generated vectors serve as pseudo CC samples, which are further used in the representation alignment step discussed in Section III-D.

D. Representation Augmentation

Although Bit Diffusion can generate pseudo CC features that are structurally close to real CC features, their feature representations may still suffer from semantic drift or distributional mismatch in the latent space. For instance, these generated CC features may overfit local noise patterns or fail to capture essential behavioral traits that characterize true CC features. To ensure that the generated features are semantically meaningful and distributionally aligned, we introduce a representation augmentation module as the final stage of our pipeline.

We design a lightweight MLP to project both real and generated features into a shared latent embedding space. Let $X = \{x_i\}_{i=1}^n$ denote the set of real CC features, and $Y = \{x'_j\}_{j=1}^m$ the set of generated pseudo CC features. Each input vector $x \in R^{158}$ is mapped to a 64-dimensional embedding $\phi(x) \in R^{64}$ through the MLP:

$$\phi(x) = \text{MLP}(x) = \text{ReLU}(W_2 \cdot \text{ReLU}(W_1 x + b_1) + b_2) \tag{5}$$

Here, $W_1 \in R^{h \times 158}$ and $W_2 \in R^{64 \times h}$ are the weight matrices of the two fully connected layers, $b_1 \in R^h$ and $b_2 \in R^{64}$ are the corresponding biases, and h is the hidden dimension. The ReLU activation introduces non-linearity and improves representation capacity.

To align the distributions of X and Y in the latent space, we use the MMD loss. MMD is a non-parametric metric that measures the distance between two distributions based on their means in a reproducing *kernel hilbert space (RKHS)*. The MMD loss is defined as:

$$\begin{aligned}\mathcal{L}_{\text{MMD}}(X,Y) =& \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} k(x_i,x_j) + \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m} k(y_i,y_j) \\ &- \frac{2}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m} k(x_i,y_j)\end{aligned} \tag{6}$$

where $k(x, y)$ is the kernel function that measures similarity between two vectors. We use a Gaussian RBF kernel defined as:

$$k(x,y) = \exp\left(-\frac{\|x-y\|^2}{2\sigma^2}\right) \tag{7}$$

where σ is a bandwidth hyperparameter controlling the kernel sensitivity.

This alignment step serves as the final stage in our framework and enables the generation of high-quality CC feature vectors suitable for downstream analysis or augmentation tasks.

IV. EXPERIMENTS SETTING

In this section, we present the experimental design for evaluating ***DEFACC***. Our primary goal is to examine whether the enhanced CC features by ***DEFACC*** can improve the performance of existing CC detection methods and the downstream FL task. Thus, we design the following research questions to evaluate ***DEFACC***:

RQ1: How does ***DEFACC*** perform in CC detection compared with original CC detection methods and other feature augmentation methods?

RQ2: How does the feature alignment module impact ***DEFACC***'s performance?

RQ3: Can ***DEFACC*** improve the performance of FL?

A. Dataset

We use the widely adopted Defects4J V1.4 benchmark dataset [59] to evaluate the effectiveness of ***DEFACC***. Defects4J provides real-world Java programs with well-documented bugs and test suites. The details of the dataset are shown in Table II. The collected dataset contains a mix of failing, genuine passing, and CC test cases, from which we construct the training and evaluation sets for CC detection experiments. To ensure the reliability of evaluation, we adopt 10-fold cross-validation, where the dataset is split into ten equal parts. The nine folds are used for training and the remaining fold is used for validation. Notably, ***DEFACC*** is a feature augmentation framework, and the 10-fold cross-validation is applied only to ***DEFACC*** to learn robust feature representations. Downstream CC detection methods are directly combined with ***DEFACC***. For FL, the downstream methods operate on datasets processed by the CC detection methods, where CC test cases are already identified and handled. Hence, the 10-fold cross-validation does not introduce unfair leakage in evaluating ***DEFACC***'s impact on FL.

TABLE II: The details of Defects4J

Program	Versions	LoC(k)	Tests	Description
Chart	26	96	3,565	Java chart library
Closure	133	90	4,457	Closure compiler
Lang	65	22	4,589	Apache commons-lang
Math	106	85	12,089	Apache commons-math
Mockito	38	23	20,200	Mocking framework for Java
Time	27	28	50,933	Standard date and time library
Total	**395**	**388**	**333,018**	**-**

B. Baselines

To answer RQ1, we compare ***DEFACC*** with four recent feature-based CC detection methods: NeuralCCD [11], MLCCI [14], TriCoCo [12], and CORE [6]. These baselines represent state-of-the-art approaches and cover diverse model architectures, including ensemble learning, triplet networks, and deep semantic modeling.

(1) **NeuralCCD** [11]: Using a MLP and ensemble system through the three types of features to detect CC test cases.
(2) **MLCCI** [14]: Extracts 60-dimensional features and uses random forest for CC detection.
(3) **TriCoCo** [12]: Identifies genuine passing tests via voting strategy with coverage and expert features, then uses a triplet network to learn deep semantic features for CC detection.
(4) **CORE** [6]: Combines three types of features via a hierarchical network for CC detection.

In addition to the origin methods, we also compare ***DEFACC*** with five feature augmentation methods:

(1) **Resampling**: Replicates minority class samples to balance the dataset.
(2) **Undersampling**: Removes majority class samples to balance class distribution.
(3) **SMOTE** [60]: Generates synthetic minority class samples by interpolating between existing minority samples and their k-nearest neighbors in feature space.
(4) **CGAN** [61]: Uses conditional generative adversarial networks to synthesize realistic minority samples.
(5) **CVAE** [62]: Employs conditional variational autoencoders to generate diverse synthetic samples.

To analyze RQ2, we further conduct ablation studies on ***DEFACC*** by removing the feature alignment component.

To assess ***DEFACC***'s effect on FL (RQ3), we follow prior work [9], [12], [63] and adopt DStar [55], Ochiai [53], Barinel [54], MLP-FL [64], CNN-FL [65], and RNN-FL [52] six FL techniques as the baseline. We relabel the identified CC tests from passing to failing when evaluating the FL effectiveness of ***DEFACC***.

C. Evaluation Metrics

We use widely adopted metrics to evaluate CC detection [2], [9], [49], [66]–[68] and FL [63], [69]–[73] effectiveness.

For CC detection (RQ1, RQ2):

(1) **Recall:** Measures the proportion of actual CC test cases that are correctly identified:

$$\text{Recall} = \frac{|T_{cc} \cap T'_{cc}|}{|T_{cc}|} \tag{8}$$

where T_{cc} denotes the set of actual CC test cases, and T'_{cc} the set identified as CC.

(2) **Precision:** Measures the proportion of identified CC test cases that are actually correct:

$$\text{Precision} = \frac{|T_{cc} \cap T'_{cc}|}{|T'_{cc}|} \tag{9}$$

(3) **F1-score:** The harmonic mean of recall and precision, which balances both:

$$F1 = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{10}$$

For FL effectiveness (RQ3):

(1) **Top-K:** Whether at least one faulty statement appears in the top K suspicious ranks, in this paper, $K \in \{1, 3, 5\}$.
(2) **Mean First Rank (MFR):** Rank of the first correctly identified faulty statement.
(3) **Mean Average Rank (MAR):** Average rank of all faulty statements.
(4) **Wilcoxon Signed-Rank Test (WSR):** Statistical significance test between baseline and ***DEFACC*** enhanced results, following prior studies [12] with p = 0.05.

D. Implementation Details

We use empirically validated settings based on prior work and expert knowledge: The binarization threshold τ is set to 0 after mean-variance normalization, to provide an unbiased separation between high and low feature values. The CNN for coverage features follows a standard design: two convolutional layers, two max-pooling layers, and one fully connected layer,

TABLE III: The results of Recall, Precision, and F1-score by comparison of original method and ***DEFACC***.

Program	Metric	Method	NeuralCCD	MLCCI	CORE	TriCoCo	Program	Metric	Method	NeuralCCD	MLCCI	CORE	TriCoCo
Chart	recall	original	32%	59%	86%	59%	Math	recall	original	16%	50%	87%	30%
		DEFACC	**47%**	**68%**	**88%**	**63%**			***DEFACC***	**28%**	**57%**	**94%**	**38%**
	Precision	original	67%	70%	76%	90%		Precision	original	60%	52%	64%	93%
		DEFACC	**73%**	**81%**	**85%**	**91%**			***DEFACC***	**72%**	**62%**	**66%**	**96%**
	F1	original	43%	64%	81%	71%		F1	original	25%	51%	74%	46%
		DEFACC	**57%**	**74%**	**86%**	**74%**			***DEFACC***	**40%**	**59%**	**77%**	**54%**
Closure	recall	original	28%	36%	**86%**	35%	Mockito	recall	original	13%	55%	79%	**32%**
		DEFACC	**41%**	**49%**	84%	**43%**			***DEFACC***	**21%**	**60%**	**84%**	30%
	Precision	original	66%	67%	80%	76%		Precision	original	47%	56%	86%	72%
		DEFACC	**78%**	**71%**	**82%**	**79%**			***DEFACC***	**60%**	**63%**	**89%**	**76%**
	F1	original	39%	47%	**83%**	48%		F1	original	20%	55%	82%	**44%**
		DEFACC	**53%**	**57%**	**83%**	**55%**			***DEFACC***	**31%**	**61%**	**86%**	43%
Lang	recall	original	19%	73%	76%	48%	Time	recall	original	30%	47%	83%	63%
		DEFACC	**30%**	**80%**	**78%**	**54%**			***DEFACC***	**42%**	**56%**	**88%**	**70%**
	Precision	original	93%	50%	85%	96%		Precision	original	35%	50%	86%	55%
		DEFACC	**96%**	**55%**	**89%**	**97%**			***DEFACC***	**48%**	**61%**	**91%**	**61%**
	F1	original	32%	59%	80%	64%		F1	original	32%	48%	75%	58%
		DEFACC	**45%**	**65%**	**83%**	**69%**			***DEFACC***	**44%**	**58%**	**89%**	**65%**

based on techniques in [52]. The MMD kernel bandwidth σ is set to 1.0 for stability. The MLP used for alignment has a hidden layer of size 128 to balance expressiveness and efficiency.

All implementations and data are publicly available at [57].

V. EXPERIMENTAL RESULTS

A. RQ1: How does DEFACC perform in CC detection compared with original CC detection methods and other feature augmentation methods?

This research question evaluates the effectiveness of ***DEFACC*** on CC detection and compares it against both state-of-the-art CC detection methods and widely used data augmentation techniques.

a) Comparison with original CC detection Methods: Table III shows the result that compares ***DEFACC*** with four representative original CC detection methods: NeuralCCD [11], MLCCI [14], TriCoCo [12], and CORE [6]. The evaluation metrics include Recall, Precision, and F1-score. Overall, ***DEFACC*** consistently outperforms all baselines in most programs and metrics. Its results are particularly strong and stable in Precision and F1-score.

On the *Math* program, ***DEFACC*** improves the Recall and F1-score of the best-performing baseline (TriCoCo) to 94% and 77%, respectively. It also raises the Precision of CORE to 96%, which is the highest among all methods. On *Closure*, although ***DEFACC***'s Recall (84%) is slightly below that of CORE (86%), it achieves higher Precision (82%) and F1-score (83%). This result suggests a better balance between Recall and Precision.

b) Comparison with Feature Augmentation Techniques: We compare ***DEFACC*** with five widely used data augmentation methods: resampling, undersampling, SMOTE [60], CGAN and CVAE. Table IV reports the results across six real-world programs and four CC detection models. Overall, ***DEFACC*** achieves the highest or second-highest scores in most settings.

Under F1-score metric, there are 24 comparisons (i.e., four CC detection methods * six programs). In 22 out of 24 comparisons, ***DEFACC*** delivers the best or tied-best. For example, on the *Math* program with TriCoCo, the undersampling obtains 66%, while the F1-score of ***DEFACC*** is 77%. On *Closure*, F1-score of resampling on NeuralCCD is 41%, the value of ***DEFACC*** is 53% (12% higher than resampling). On *Lang*, it outperforms all other augmentation methods across all CC detection methods. These results demonstrate that ***DEFACC*** provides consistent gains and strong generalization in CC detection.

There are two exceptions where resampling slightly outperforms ***DEFACC***. On the *Closure* program with TriCoCo, resampling achieves an F1 of 86%, compared to ***DEFACC***'s 83%. On *Mockito* with CORE, resampling reaches 46%, while ***DEFACC*** achieves 43%. These results are reasonable. Resampling uses high-confidence real samples, which are clean and semantically accurate. In contrast, ***DEFACC*** generates pseudo-CC features, which, although aligned, may introduce slight noise and lead to marginal drops in precision.

Undersampling performs the worst in nearly all cases. It removes a large portion of the majority class, which causes significant information loss. As a result, models trained on undersampled data struggle to capture CC patterns and perform poorly.

SMOTE performs slightly better than undersampling but worse than ***DEFACC*** and resampling. SMOTE generates new samples by interpolating between existing minority examples. However, this process may fail to preserve the complex semantics of CC features, especially in high-dimensional spaces that combine expert, coverage, and semantic signals. The result is reduced discriminability and less robust classification.

In summary, resampling performs well in precision-focused scenarios but lacks diversity. ***DEFACC*** offers a better trade-off between Recall and Precision. By generating diverse yet semantically consistent CC features, ***DEFACC*** proves more effective in imbalanced and feature-scarce CC detection tasks.

Answer to RQ1: DEFACC outperforms both original CC detection methods and general feature augmentation techniques. Compared to existing detection methods, ***DEFACC*** achieves an average improvement of 7.12%, 6.25%, and 7.79% on Recall, Precision, and F1-score, respectively. Compared to resampling, undersampling, and SMOTE, ***DEFACC*** further improves the Recall by up to 5.63% on average.

TABLE IV: The results of Recall, Precision, and F1-score by comparison of augmentation method and ***DEFACC***.

Program	Metric	Method	NeuralCCD	MLCCI	CORE	TriCoCo	Program	Metric	Method	NeuralCCD	MLCCI	CORE	TriCoCo
Chart	Recall	re	33%	60%	**88%**	61%	Math	Recall	re	16%	52%	90%	30%
		under	27%	52%	74%	52%			under	13%	43%	77%	26%
		smote	33%	56%	81%	61%			smote	15%	52%	82%	28%
		CGAN	38%	59%	84%	58%			CGAN	22%	54%	91%	33%
		CVAE	40%	60%	85%	59%			CVAE	23%	55%	93%	34%
		DEFACC	**47%**	**68%**	**88%**	**63%**			DEFACC	**28%**	**57%**	**94%**	**38%**
	Precision	re	70%	72%	78%	**93%**		Precision	re	62%	54%	66%	**97%**
		under	57%	59%	65%	79%			under	51%	45%	57%	81%
		smote	63%	66%	72%	90%			smote	63%	49%	65%	95%
		CGAN	67%	70%	79%	88%			CGAN	70%	58%	67%	95%
		CVAE	66%	71%	80%	89%			CVAE	68%	56%	**68%**	95%
		DEFACC	**73%**	**81%**	**85%**	91%			DEFACC	**72%**	**62%**	66%	96%
	F1	re	45%	65%	83%	**74%**		F1	re	25%	53%	76%	46%
		under	37%	55%	69%	63%			under	21%	44%	66%	39%
		smote	41%	61%	76%	73%			smote	24%	50%	73%	43%
		CGAN	48%	64%	81%	70%			CGAN	32%	57%	76%	52%
		CVAE	50%	65%	82%	71%			CVAE	34%	55%	**78%**	50%
		DEFACC	**57%**	**74%**	**86%**	**74%**			DEFACC	**40%**	**59%**	77%	**54%**
Closure	Recall	re	21%	**49%**	78%	33%	Mockito	Recall	re	13%	52%	**88%**	29%
		under	18%	42%	65%	30%			under	12%	46%	74%	24%
		smote	20%	45%	71%	32%			smote	12%	49%	79%	27%
		CGAN	30%	45%	84%	37%			CGAN	16%	57%	87%	32%
		CVAE	32%	46%	**86%**	38%			CVAE	17%	55%	**88%**	**34%**
		DEFACC	**41%**	**49%**	84%	**43%**			DEFACC	**21%**	**60%**	84%	30%
	Precision	re	65%	58%	70%	**89%**		Precision	re	60%	53%	67%	73%
		under	54%	50%	59%	76%			under	49%	44%	56%	60%
		smote	59%	54%	63%	86%			smote	61%	48%	62%	74%
		CGAN	69%	68%	80%	85%			CGAN	62%	59%	84%	74%
		CVAE	71%	69%	81%	88%			CVAE	**64%**	60%	83%	75%
		DEFACC	**78%**	**71%**	**82%**	79%			DEFACC	60%	**63%**	**89%**	**76%**
	F1	re	32%	53%	70%	48%		F1	re	21%	52%	77%	41%
		under	26%	45%	62%	43%			under	19%	45%	63%	34%
		smote	29%	49%	67%	46%			smote	21%	48%	70%	40%
		CGAN	42%	54%	82%	50%			CGAN	28%	56%	82%	45%
		CVAE	44%	55%	83%	53%			CVAE	27%	57%	85%	**46%**
		DEFACC	**53%**	**57%**	**83%**	**55%**			DEFACC	**31%**	**61%**	**86%**	43%
Lang	Recall	re	18%	47%	**85%**	27%	Time	Recall	re	34%	51%	84%	64%
		under	15%	41%	72%	23%			under	19%	46%	73%	30%
		smote	17%	44%	78%	26%			smote	33%	49%	79%	60%
		CGAN	27%	76%	79%	50%			CGAN	39%	54%	87%	67%
		CVAE	24%	78%	80%	51%			CVAE	36%	53%	86%	65%
		DEFACC	**30%**	**80%**	78%	**54%**			DEFACC	**42%**	**56%**	**88%**	**70%**
	Precision	re	61%	**55%**	71%	92%		Precision	re	40%	56%	69%	57%
		under	49%	47%	60%	76%			under	35%	48%	58%	41%
		smote	55%	51%	65%	88%			smote	42%	52%	63%	59%
		CGAN	94%	54%	85%	95%			CGAN	42%	57%	87%	67%
		CVAE	95%	**55%**	86%	96%			CVAE	44%	58%	88%	59%
		DEFACC	**96%**	**55%**	**89%**	**97%**			DEFACC	**48%**	**61%**	**91%**	**61%**
	F1	re	28%	51%	77%	41%		F1	re	36%	53%	77%	60%
		under	22%	44%	65%	34%			under	25%	47%	65%	35%
		smote	25%	47%	71%	39%			smote	37%	50%	71%	59%
		CGAN	36%	63%	81%	65%			CGAN	42%	57%	86%	63%
		CVAE	38%	64%	82%	67%			CVAE	40%	55%	87%	62%
		DEFACC	**45%**	**65%**	**83%**	**69%**			DEFACC	**44%**	**58%**	**89%**	**65%**

're' means resampling and 'under' means undersampling.

*B. RQ2: How does the feature alignment module impact **DEFACC**'s performance?*

To evaluate the effectiveness of the feature alignment module, we conduct an ablation study. We compare the ***DEFACC*** with a variant that removes the alignment component, referred to as ***DEFACC***/wo fa. Table V presents the results on six real-world programs.

TABLE V: The ablation results.

Program	Method	Recall	Precision	F1
Chart	***DEFACC***	88%	85%	86%
	DEFACC/wo fa	83%	80%	81%
Closure	***DEFACC***	84%	82%	83%
	DEFACC/wo fa	78%	80%	79%
Lang	***DEFACC***	78%	89%	83%
	DEFACC/wo fa	75%	85%	80%
Math	***DEFACC***	94%	66%	77%
	DEFACC/wo fa	90%	61%	73%
Mockito	***DEFACC***	84%	89%	86%
	DEFACC/wo fa	81%	84%	83%
Time	***DEFACC***	88%	91%	89%
	DEFACC/wo fa	85%	87%	86%

Removing the alignment module consistently reduces performance across all datasets and metrics, including Recall, Precision, and F1-score. On average, ***DEFACC*** achieves 3% to 5% higher F1-scores than its variant. For example, on the *Math* dataset, the F1-score drops from 77% to 73% without the alignment module. Similar gaps appear on *Chart* (86% vs. 81%) and *Closure* (83% vs. 79%).

Answer to RQ2: The feature alignment module plays a critical role in enhancing ***DEFACC***'s effectiveness. Across six real-world programs, incorporating the alignment module yields an average improvement of **4.3% in F1-score**, compared to the variant without it. These improvements confirm that aligning the distribution of real and generated features strengthens semantic consistency and leads to more accurate CC detection.

C. RQ3: Can DEFACC improve the performance of FL?

To evaluate whether ***DEFACC*** benefits downstream FL, we apply it to six representative FL techniques. We integrate ***DEFACC*** with TriCoCo and compare its performance against the original TriCoCo (without ***DEFACC***) across the six FL techniques. For each method, we compare the results on two settings: (1) the original setting, where tests include unfiltered CC tests, and (2) the CC handling setting, where tests identified as CC are relabeled from `pass` (0) to `fail` (1).

Table VI shows the results under TriCoCo with original method and different data imbalance handling methods. Across all six FL methods, ***DEFACC*** consistently improves localization performance. For example, on statistical methods like Dstar, ***DEFACC*** raises TOP-1 from 18 to 24 and reduces MFR from 220.20 to 185.34. For learning-based FL methods, ***DEFACC*** consistently outperforms the original baselines. For example, with MLP-FL, the number of correctly localized faults in TOP-1, TOP-3, and TOP-5 increases from 12, 24, and 30 to 16, 32, and 38, respectively. This corresponds to relative improvements of 33%, 33%, and 27%. Similar trends are observed in CNN-FL and RNN-FL, indicating that correcting CC labels helps neural models better prioritize fault-revealing statements.

TABLE VI: The results of Top-1, Top-3, Top-5, MFR and MAR under TriCoCo with original method and different data imbalance handling methods.

Metric	Method	Dstar	Ochiai	Barinel	MLP-FL	CNN-FL	RNN-FL
Top-1	original	18	17	16	12	15	12
	re	17	18	15	12	14	12
	under	14	15	13	10	12	10
	smote	16	16	14	11	13	11
	CGAN	20	19	18	14	18	15
	CVAE	21	20	19	14	18	15
	DEFACC	**24**	**22**	**20**	**16**	**20**	**17**
Top-3	original	35	35	29	24	33	25
	re	34	34	28	23	32	24
	under	29	28	25	20	27	21
	smote	32	31	26	22	29	23
	CGAN	38	37	33	29	36	30
	CVAE	39	38	34	30	36	30
	DEFACC	**42**	**40**	**36**	**32**	**39**	**33**
Top-5	original	52	53	46	30	53	33
	re	50	51	44	29	51	32
	under	44	45	40	26	45	28
	smote	48	49	42	28	49	30
	CGAN	56	57	49	36	56	38
	CVAE	57	58	50	37	57	39
	DEFACC	**60**	**58**	**52**	**38**	**60**	**42**
MFR	original	220.20	241.74	258.56	401.23	343.12	411.95
	re	230.12	251.23	268.77	412.10	353.45	423.32
	under	270.34	291.56	308.12	452.33	393.12	463.22
	smote	250.23	271.23	288.45	432.12	373.56	443.10
	CGAN	200.45	221.78	238.67	383.12	323.12	393.22
	CVAE	195.56	216.89	243.12	378.45	318.77	388.99
	DEFACC	**185.34**	**210.66**	**235.42**	**360.89**	**298.77**	**371.55**
MAR	original	512.61	531.59	587.96	653.29	594.19	725.78
	re	522.33	542.10	598.22	664.33	604.11	737.45
	under	562.11	582.45	638.55	704.55	644.78	777.90
	smote	542.10	562.33	618.44	684.34	624.56	757.33
	CGAN	492.45	511.78	568.90	634.45	574.33	707.89
	CVAE	482.78	521.22	570.34	624.56	563.77	697.45
	DEFACC	**472.33**	**508.12**	**559.48**	**614.25**	**558.02**	**693.10**

TABLE VII: Statistical comparison of DEFACC and the original FL approaches.

Comparison	greater	less	two-sided	conclusion
Dstar(***DEFACC***) v.s. Dstar	1.00E+00	3.05E-05	6.10E-05	**BETTER**
Ochiai(***DEFACC***) v.s. Ochiai	1.00E+00	1.22E-04	2.21E-03	**BETTER**
Barinel(***DEFACC***) v.s. Barinel	1.00E+00	3.71E-04	2.29E-03	**BETTER**
MLP-FL(***DEFACC***) v.s. MLP-FL	9.99E-01	1.10E-03	2.61E-03	**BETTER**
CNN-FL(***DEFACC***) v.s. CNN-FL	9.52E-01	4.85E-02	9.69E-02	**BETTER**
RNN-FL(***DEFACC***) v.s. RNN-FL	9.93E-01	7.41E-03	1.48E-02	**BETTER**

We also conduct the Wilcoxon Signed-Rank (WSR) test to validate statistical significance. As shown in Table VII, all p-values are below 0.05, confirming that the improvements brought by ***DEFACC*** are statistically significant. For instance, Dstar and Ochiai show two-sided p-values of 6.10×10^{-5} and 2.21×10^{-3} respectively, while learning-based methods such as RNN-FL also yield $p < 0.02$.

To further support this conclusion, we analyze the improvement in terms of MAR/LoC value for a faulty version, where 'LoC' refers to the number of executable lines. Fig. 4 visualizes the MAR/LoC distributions for six FL techniques, comparing the original TriCoCo and ***DEFACC***-enhanced versions. The results show that ***DEFACC*** consistently reduces MAR/LoC across all methods. This effect is especially notable for deep learning models such as RNN-FL, which benefit more from noise reduction. By correcting CC labels, ***DEFACC*** removes misleading signals and helps models focus on fault-relevant patterns, ultimately improving both accuracy and robustness.

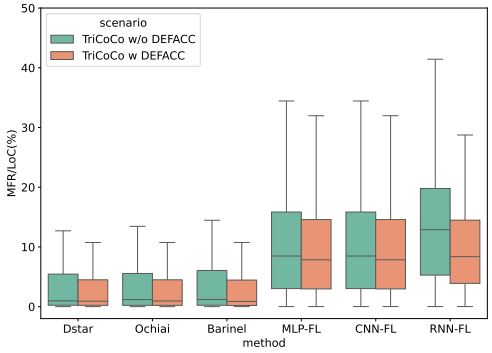


Fig. 4: The boxplot of original FL methods and these with ***DEFACC***.

Answer to RQ3: DEFACC improves FL performance across all baselines under both value-based and statistical-based metrics. These results confirm that correcting CC labels enhances the accuracy and robustness of FL.

VI. Discussion

A. Why does *DEFACC* not perform well on some programs?

DEFACC performs well on several programs, mainly due to its diffusion module, which captures latent CC feature structures and generates semantically consistent pseudo-CC features. These features help alleviate class imbalance and enhance the model's learning process. In addition, the feature alignment module reduces the distribution gap between real and generated features using MMD loss, further improving performance. For example, on the *Chart* program, it improves the F1-score of NeuralCCD from 43% to 57%, and recall from 32% to 47%.

However, on programs like *Closure* and *Mockito*, the improvements are limited in some metrics. For instance, in *Mockito*, the F1-score of ***DEFACC*** with CORE reaches 83%, slightly lower than the original CORE's 86%. This reduced performance is likely caused by noisy or complex CC distributions in these datasets, which make it harder for the diffusion module to learn precise feature boundaries. In such cases, the MMD-based alignment may also fail to fully close the gap between real and generated features. Even so, ***DEFACC*** still ranks among the top two or three methods across most metrics

and datasets. Overall, it is acceptable since ***DEFACC*** performs baselines in general.

*B. What are the specific contributions of the **DEFACC**?*

The diffusion module learns the distribution of CC features in sparse binary space and generates pseudo-CC features. This increases data diversity. Experimental results show that generated features resemble real CC features in structure and semantics. The feature alignment module uses MMD loss to project both real and generated features into a shared latent space, reducing semantic drift. Ablation studies confirm that removing this module degrades recall, precision, and F1 score. Together, these two modules work in tandem: the diffusion module enriches training data, while the alignment module ensures distributional consistency. This synergy improves the accuracy and robustness of ***DEFACC*** in CC detection and FL.

*C. What advantages does **DEFACC** offer over traditional CC detection methods?*

Traditional CC detection approaches, including heuristics and machine learning models, often operate on imbalanced datasets, where CC test cases are rare. This limits the model capacity to learn useful features and increases misclassification. ***DEFACC*** mitigates this issue through generative feature-level augmentation. Its diffusion module synthesizes new CC features, while the alignment module ensures their semantic and distributional alignment with real features. This results in more effective model training. Experiments show that ***DEFACC*** consistently outperforms traditional methods across multiple metrics in both CC detection and FL tasks.

Beyond performance gains, another key advantage of ***DEFACC*** is its compatibility with diverse CC detection methods. Unlike standalone detectors, ***DEFACC*** functions as a plug-in feature augmentation framework that integrates seamlessly into existing workflows. For each detector (e.g., NeuralCCD, MLCCI, CORE, TriCoCo), we preserve its native feature extraction process and train the diffusion model directly in that feature space. ***DEFACC*** then generates pseudo-CC samples aligned via MMD to the real CC distribution and augments the detector's training data without modifying its architecture or inference pipeline. If detectors share the same feature representation, the same trained ***DEFACC*** instance can be reused; otherwise, separate instances are trained for different feature spaces. This flexible integration ensures that ***DEFACC*** can enhance a wide range of CC detectors while maintaining full compatibility with their original designs.

*D. Effectiveness of **DEFACC** with Small Training Sets*

To investigate whether ***DEFACC*** remains effective under limited training data, we conduct experiments using 30%, 50%, and 70% of the original training set. Table VIII shows the experimental results. ***DEFACC*** consistently improves performance under reduced data, especially for weaker baselines (e.g., NeuralCCD F1 improves from 32% to 44% at 70%). Stronger methods, such as CORE and TriCoCo, also gain stable increases. These results confirm that ***DEFACC*** remains effective and robust even with limited training sets.

TABLE VIII: DEFACC performance with small training sets (30%, 50%, 70%) across different CC detection methods.

metric	method	ori	30%	50%	70%	100%
Rcall	NeuralCCD	23%	27%	30%	32%	35%
	MLCCI	53%	56%	58%	60%	62%
	CORE	83%	83%	84%	86%	86%
	TriCoCo	45%	47%	48%	49%	50%
Precision	NeuralCCD	61%	64%	67%	69%	71%
	MLCCI	58%	61%	63%	65%	66%
	CORE	80%	80%	83%	83%	84%
	TriCoCo	80%	80%	83%	83%	83%
F1	NeuralCCD	32%	33%	38%	44%	45%
	MLCCI	54%	56%	58%	61%	62%
	CORE	79%	80%	81%	83%	84%
	TriCoCo	55%	57%	58%	59%	60%

E. Threats to Validity

Dataset-related Threats. The Defects4J dataset may lack certain CC test case types, limiting generalization. Labeling errors can introduce noise into training, leading to poor feature learning and unreliable evaluation.

Model-related Threats. ***DEFACC*** assumes certain data distributions for bit-level diffusion. If real-world data violates these assumptions, generated features may exhibit semantic drift. MMD alignment may not fully compensate for such a mismatch.

Experimental Setup Threats. Although 10-fold cross-validation provides a reasonable estimate of model performance, it is still subject to randomness and sampling bias. These factors may lead to performance fluctuations across different data splits. As a result, the evaluation may not fully reflect the model's robustness or the general effectiveness of ***DEFACC***.

VII. Conclusion

In this paper, we propose ***DEFACC***, a diffusion-enhanced and feature-aligned framework for CC detection. ***DEFACC*** leverages a combination of expert features, coverage features, and semantic features, and employs both feature augmentation and representation alignment to produce high-quality pseudo CC features. For feature augmentation part, we use a bit diffusion module that learns the distribution of CC features in a sparse binary space and generates pseudo-CC features. For representation alignment part, we adopt MMD loss to align the representation between the real and generated features.

We conduct extensive experiments on the Defects4J benchmark dataset. The results demonstrate that ***DEFACC*** significantly improves the representation quality of CC in the training set, leading to notable performance gains in downstream CC detection and FL. For example, when augmented with ***DEFACC***, the four CC detection models achieve consistent improvements across multiple evaluation metrics, including Recall, Precsision, and F1-score. Experimental results reveal that the bit diffusion model effectively captures the latent structure of CC features, while the embedding alignment mechanism successfully mitigates distributional shifts between real and generated features.

REFERENCES

[1] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, 2009, pp. 1–5.

[2] W. Masri and R. Abou Assi, "Cleansing test suites from coincidental correctness to enhance fault-localization," in *2010 third international conference on software testing, verification and validation*. IEEE, 2010, pp. 165–174.

[3] R. Abou Assi, C. Trad, M. Maalouf, and W. Masri, "Coincidental correctness in the defects4j benchmark," *Software Testing, Verification and Reliability*, vol. 29, no. 3, p. e1696, 2019.

[4] R. Abou Assi, W. Masri, and C. Trad, "How detrimental is coincidental correctness to coverage-based fault detection and localization? an empirical study," *Software Testing, Verification and Reliability*, vol. 31, no. 5, p. e1762, 2021.

[5] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM transactions on software engineering and methodology (TOSEM)*, vol. 23, no. 1, pp. 1–28, 2014.

[6] H. Xie, Y. Lei, M. Li, M. Yan, and S. Zhang, "Combining coverage and expert features with semantic representation for coincidental correctness detection," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1770–1782.

[7] A. Bandyopadhyay, "Mitigating the effect of coincidental correctness in spectrum based fault localization," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 479–482.

[8] S. Dass, X. Xue, and A. S. Namin, "Ensemble random forests classifier for detecting coincidentally correct test cases," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 1326–1331.

[9] J. Yu, Y. Lei, H. Xie, L. Fu, and C. Liu, "Context-based cluster fault localization," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 482–493.

[10] S. Dass, X. Xue, and A. Siami Namin, "Detection of coincidentally correct test cases through random forests," *arXiv e-prints*, pp. arXiv–2006, 2020.

[11] Z. Tao, Y. Lei, H. Xie, and J. Hu, "Neuralccd: Integrating multiple features for neural coincidental correctness detection," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 85–96.

[12] H. Xie, Y. Lei, M. Yan, S. Li, X. Mao, Y. Yu, and D. Lo, "Towards more precise coincidental correctness detection with deep semantic learning," *IEEE Transactions on Software Engineering*, 2024.

[13] A. Sabbaghi, M. R. Keyvanpour, and S. Parsa, "Fcci: A fuzzy expert system for identifying coincidental correct test cases," *Journal of Systems and Software*, vol. 168, p. 110635, 2020.

[14] Y. Wu, S. Tian, Z. Yang, Z. Li, Y. Liu, and X. Chen, "Identifying coincidental correct test cases with multiple features extraction for fault localization," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2023, pp. 800–809.

[15] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in neural information processing systems*, vol. 33, pp. 6840–6851, 2020.

[16] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep unsupervised learning using nonequilibrium thermodynamics," in *International conference on machine learning*. pmlr, 2015, pp. 2256–2265.

[17] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole, "Score-based generative modeling through stochastic differential equations," *arXiv preprint arXiv:2011.13456*, 2020.

[18] T. Chen, R. Zhang, and G. Hinton, "Analog bits: Generating discrete data using diffusion models with self-conditioning," *arXiv preprint arXiv:2208.04202*, 2022.

[19] P. Dhariwal and A. Nichol, "Diffusion models beat gans on image synthesis," *Advances in neural information processing systems*, vol. 34, pp. 8780–8794, 2021.

[20] Y. Song and S. Ermon, "Generative modeling by estimating gradients of the data distribution," *Advances in neural information processing systems*, vol. 32, 2019.

[21] J. Ho, T. Salimans, A. Gritsenko, W. Chan, M. Norouzi, and D. J. Fleet, "Video diffusion models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 8633–8646, 2022.

[22] O. Bar-Tal, H. Chefer, O. Tov, C. Herrmann, R. Paiss, S. Zada, A. Ephrat, J. Hur, G. Liu, A. Raj *et al.*, "Lumiere: A space-time diffusion model for video generation," in *SIGGRAPH Asia 2024 Conference Papers*, 2024, pp. 1–11.

[23] Z. Luo, D. Chen, Y. Zhang, Y. Huang, L. Wang, Y. Shen, D. Zhao, J. Zhou, and T. Tan, "Videofusion: Decomposed diffusion models for high-quality video generation," *arXiv preprint arXiv:2303.08320*, 2023.

[24] Z. Xing, Q. Feng, H. Chen, Q. Dai, H. Hu, H. Xu, Z. Wu, and Y.-G. Jiang, "A survey on video diffusion models," *ACM Computing Surveys*, vol. 57, no. 2, pp. 1–42, 2024.

[25] D. Zhou, W. Wang, H. Yan, W. Lv, Y. Zhu, and J. Feng, "Magicvideo: Efficient video generation with latent diffusion models," *arXiv preprint arXiv:2211.11018*, 2022.

[26] Y. Wang, X. Chen, X. Ma, S. Zhou, Z. Huang, Y. Wang, C. Yang, Y. He, J. Yu, P. Yang *et al.*, "Lavie: High-quality video generation with cascaded latent diffusion models," *International Journal of Computer Vision*, vol. 133, no. 5, pp. 3059–3078, 2025.

[27] J. Austin, D. D. Johnson, J. Ho, D. Tarlow, and R. Van Den Berg, "Structured denoising diffusion models in discrete state-spaces," *Advances in neural information processing systems*, vol. 34, pp. 17 981–17 993, 2021.

[28] E. Hoogeboom, D. Nielsen, P. Jaini, P. Forré, and M. Welling, "Argmax flows and multinomial diffusion: Learning categorical distributions," *Advances in neural information processing systems*, vol. 34, pp. 12 454–12 465, 2021.

[29] X. Li, J. Thickstun, I. Gulrajani, P. S. Liang, and T. B. Hashimoto, "Diffusion-lm improves controllable text generation," *Advances in neural information processing systems*, vol. 35, pp. 4328–4343, 2022.

[30] P. Yu, S. Xie, X. Ma, B. Jia, B. Pang, R. Gao, Y. Zhu, S.-C. Zhu, and Y. N. Wu, "Latent diffusion energy-based model for interpretable text modeling," *arXiv preprint arXiv:2206.05895*, 2022.

[31] L. Yang, Z. Zhang, Y. Song, S. Hong, R. Xu, Y. Zhao, W. Zhang, B. Cui, and M.-H. Yang, "Diffusion models: A comprehensive survey of methods and applications," *ACM Computing Surveys*, vol. 56, no. 4, pp. 1–39, 2023.

[32] H. Koo and T. E. Kim, "A comprehensive survey on generative diffusion models for structured data," *arXiv preprint arXiv:2306.04139*, 2023.

[33] Z. Chen, L. Wang, Y. Wu, X. Liao, Y. Tian, and J. Zhong, "An effective deployment of diffusion lm for data augmentation in low-resource sentiment classification," *arXiv preprint arXiv:2409.03203*, 2024.

[34] H. Wang, Y. Shang, Z. Yuan, J. Wu, J. Yan, and Y. Yan, "Quest: Low-bit diffusion model quantization via efficient selective finetuning," *arXiv preprint arXiv:2402.03666*, 2024.

[35] W. Feng, H. Qin, C. Yang, Z. An, L. Huang, B. Diao, F. Wang, R. Tao, Y. Xu, and M. Magno, "Mpq-dm: Mixed precision quantization for extremely low bit diffusion models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 16, 2025, pp. 16 595–16 603.

[36] C. Ding, P. Gao, J. Li, and W. Wu, "Multi-source deep transfer learning algorithm based on feature alignment," *Artificial Intelligence Review*, vol. 56, no. Suppl 1, pp. 769–791, 2023.

[37] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[38] A. K. Menon, S. Jayasumana, A. S. Rawat, H. Jain, A. Veit, and S. Kumar, "Long-tail learning via logit adjustment," *arXiv preprint arXiv:2007.07314*, 2020.

[39] J. Ren, C. Yu, X. Ma, H. Zhao, S. Yi *et al.*, "Balanced meta-softmax for long-tailed visual recognition," *Advances in neural information processing systems*, vol. 33, pp. 4175–4186, 2020.

[40] R. Vigneswaran, M. T. Law, V. N. Balasubramanian, and M. Tapaswi, "Feature generation for long-tail classification," in *Proceedings of the twelfth Indian conference on computer vision, graphics and image processing*, 2021, pp. 1–9.

[41] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. March, and V. Lempitsky, "Domain-adversarial training of neural networks," *Journal of machine learning research*, vol. 17, no. 59, pp. 1–35, 2016.

[42] M. Long, H. Zhu, J. Wang, and M. I. Jordan, "Deep transfer learning with joint adaptation networks," in *International conference on machine learning*. PMLR, 2017, pp. 2208–2217.

[43] T. Zhang, X. Liu, L. Gong, S. Wang, X. Niu, and L. Shen, "Late fusion multiple kernel clustering with local kernel alignment maximization," *IEEE Transactions on Multimedia*, vol. 25, pp. 993–1007, 2021.

[44] T. Zhou, Y. Yuan, B. Wang, and E. Konukoglu, "Federated feature augmentation and alignment," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[45] S. Xu, H. Zhang, X. Xu, X. Hu, Y. Xu, L. Dai, K.-S. Choi, and P.-A. Heng, "Representative feature alignment for adaptive object detection," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 33, no. 2, pp. 689–700, 2022.

[46] X. Xie, Z.-H. You, S.-B. Chen, L.-L. Huang, J. Tang, and B. Luo, "Feature enhancement and alignment for oriented object detection," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 17, pp. 778–787, 2023.

[47] C.-L. Li, W.-C. Chang, Y. Cheng, Y. Yang, and B. Póczos, "Mmd gan: Towards deeper understanding of moment matching network," *Advances in neural information processing systems*, vol. 30, 2017.

[48] Y. Miao, Z. Chen, S. Li, Z. Zhao, and Y. Zhou, "A clustering-based strategy to identify coincidental correctness in fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 05, pp. 721–741, 2013.

[49] X. Xue, Y. Pang, and A. S. Namin, "Trimming test suites with coincidentally correct test cases for enhancing fault localizations," in *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 2014, pp. 239–244.

[50] M. Li, Y. Lei, H. Xie, J. Wang, C. Liu, and Z. Deng, "Contrastive coincidental correctness representation learning," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 252–263.

[51] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[52] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, "A study of effectiveness of deep learning in locating real faults," *Information and Software Technology*, vol. 131, p. 106486, 2021.

[53] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[54] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[55] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[56] E. Deza, M. M. Deza, M. M. Deza, and E. Deza, *Encyclopedia of distances*. Springer, 2009.

[57] Anonymous, "Our artifact." [Online]. Available: https://anonymous.4open.science/r/DEAFCC-3B4E

[58] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, "Methods2test: A dataset of focal methods mapped to test cases," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 299–303.

[59] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[60] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[61] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[62] A. Mishra, S. Krishna Reddy, A. Mittal, and H. A. Murthy, "A generative model for zero shot learning using conditional variational autoencoders," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2018, pp. 2188–2196.

[63] H. Xie, Y. Lei, M. Yan, Y. Yu, X. Xia, and X. Mao, "A universal data augmentation approach for fault localization," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 48–60.

[64] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Mathematical Problems in Engineering*, vol. 2016, no. 1, p. 1820454, 2016.

[65] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 445–455.

[66] F. Feyzi and S. Parsa, "A program slicing-based method for effective detection of coincidentally correct test cases," *Computing*, vol. 100, no. 9, pp. 927–969, 2018.

[67] Y. Liu, M. Li, Y. Wu, and Z. Li, "A weighted fuzzy classification approach to identify and manipulate coincidental correct test cases for fault localization," *Journal of Systems and Software*, vol. 151, pp. 20–37, 2019.

[68] F. Feyzi, "Cgt-fl: using cooperative game theory to effective fault localization in presence of coincidental correctness," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3873–3927, 2020.

[69] Y. Lei, C. Liu, H. Xie, S. Huang, M. Yan, and Z. Xu, "Bcl-fl: A data augmentation approach with between-class learning for fault localization," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 289–300.

[70] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "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*, 2021, pp. 664–676.

[71] M. N. Rafi, D. J. Kim, A. R. Chen, T.-H. Chen, and S. Wang, "Towards better graph neural network-based fault localization through enhanced code representation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1937–1959, 2024.

[72] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[73] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 196–202.