

What You See Is What You Get: Attention-based Self-guided Automatic Unit Test Generation

Xin Yin, Chao Ni*, Xiaodan Xu, Xiaohu Yang

The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China
{xyin, chaoni, xiaodanxu, yangxh}@zju.edu.cn

Abstract—Software defects heavily affect software’s functionalities and may cause huge losses. Recently, many AI-based approaches have been proposed to detect defects, which can be divided into two categories: software defect prediction and automatic unit test generation. While these approaches have made great progress in software defect detection, they still have several limitations in practical application, including the low confidence of prediction models and the inefficiency of unit testing models.

To address these limitations, we propose a WYSIWYG (i.e., What You See Is What You Get) approach: Attention-based Self-guided Automatic Unit Test GenERation (AUGER), which contains two stages: defect detection and error triggering. In the former stage, AUGER first detects the proneness of defects. Then, in the latter stage, it guides to generate unit tests for triggering such an error with the help of critical information obtained by the former stage. To evaluate the effectiveness of AUGER, we conduct a large-scale experiment by comparing with the state-of-the-art (SOTA) approaches on the widely used datasets (i.e., Bears, Bugs.jar, and Defects4J). AUGER makes great improvements by 4.7% to 35.3% and 17.7% to 40.4% in terms of F1-score and Precision in defect detection, and can trigger 23 to 84 more errors than SOTAs in unit test generation. Besides, we also conduct a further study to verify the generalization in practical usage by collecting a new dataset from real-world projects.

Index Terms—Defect, Unit Test Generation, Error-Triggering

I. INTRODUCTION

With the rapid advancement of industrial automation, code complexity and scale have increased, posing significant challenges in managing defects. Developers often miss potential defects unless explicit errors occur, reducing software quality. Recently, many approaches have been proposed to detect defects, which can be divided into two categories: software defect prediction and automatic unit test generation. Despite the advancements in these approaches, they still face certain challenges and limitations.

Current defect detection approaches provide solely binary predictions [1]–[8], indicating the presence of defects in code snippets or statements, but lack detailed explanations, making it challenging for developers to understand and trust the reliability of these predictions. To address this limitation, LineVul [9] uses attention scores for rationale, and some approaches incorporate the code’s graph structure [10]–[15]. However, these efforts only focus on explaining how models make specific decisions and neglect to provide detailed insights into the conditions that cause defects. Meanwhile, Steenhoek et al. [16] also show

significant variability among different models under identical input conditions, eroding developers’ trust in detection results.

Unit test generation approaches are crucial for ensuring software security and quality. Traditional unit test generation approaches prioritize achieving high code coverage. However, research shows that high code coverage does not always trigger errors effectively [17], [18]. Recently, learning-based generation approaches [19]–[21] have made remarkable strides in progress. However, they focus on randomly generating a large number of test cases by training on the Method2Test dataset [22], which contains numerous clean methods and non-error-triggering test cases, lacking effective information guidance. This leads to poor test efficiency, failing to efficiently trigger errors.

To address these limitations, we propose a WYSIWYG approach: Attention-based Self-guided Automatic Unit Test GenERation (AUGER), which contains two stages: defect detection and error triggering. In the first stage, AUGER detects defect proneness. In the latter stage, it guides the large language model (LLM) to generate unit tests for triggering such an error with the help of critical information obtained by the former stage. As a result, AUGER will provide developers with defect detection results, convincing error-triggering unit tests, and corresponding test results. This automated approach instills confidence in developers regarding the detection results.

To investigate the effectiveness of AUGER, we first extract and filter methods from several widely used datasets [23]–[25], resulting in a total of 40,523 defective and non-defective methods. Subsequently, we conduct comprehensive experiments to assess AUGER’s performance in defect detection. The results indicate that AUGER can achieve an F1-score of 0.276, Precision of 0.198, and PR-AUC of 0.208 on Defects4J, which improves the baselines by 11.3% to 35.3%, 20.0% to 40.4%, and 24.6% to 69.1%, respectively. Besides, we conduct extensive experiments to evaluate the effectiveness of AUGER in error triggering. The outcomes demonstrate that AUGER effectively triggers 35 and 84 errors in different scenarios, with a noticeably higher precision than the baselines. In our collection of real-world projects after March 2023, AUGER also achieves promising performance in error triggering. Our main contributions are summarized as follows:

A. WYSIWYG: Defect Detection with High Confidence: AUGER provides defect detection results, convincing error-triggering unit tests, and corresponding test results, which helps to instill greater confidence in developers.

B. Error Triggering with High Efficiency: AUGER guides the LLM to generate unit tests that trigger errors by leveraging

* This is the corresponding author
Chao Ni is also with Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute, Hangzhou, China

critical information extracted from defective code, thereby narrowing down the search space.

C. Extensive Evaluations: We evaluate AUGER against current state-of-the-art approaches on the widely studied datasets [23]–[25]. To prevent data leakage, we also collect an additional real-world dataset for evaluation. The replication package is publicly available at [26].

II. BACKGROUND AND MOTIVATION

Defect detection and unit test generation are the main approaches to ensure software quality. In this section, we aim to explore the challenges and limitations of existing defect detection approaches and unit test generation approaches.

A. Limited confidence in defect detection models

The limited confidence in defect detection models arises from two aspects: solely binary predictions and model inconsistency.

Current defect detection models often provide solely binary predictions, indicating whether a code snippet or statement contains defects or not [1]–[8], [27]. These results lack detailed explanations, making it challenging for developers to understand and trust the reliability of these predictions. LineVul [9] utilizes attention scores to explain the rationale behind the model’s decision-making. Moreover, some approaches enhance the model’s capabilities by incorporating the graph structure of code [10]–[15]. While these efforts have incorporated explanatory features, they often focus on why the model made a specific decision rather than providing detailed insights into the conditions that cause defects (e.g., the inputs and outputs that can expose defects).

Meanwhile, the inconsistency of the defect detection models also undermines developers’ trust. Steenhoek et al. [16] reveal significant variability among different models, as they may produce entirely divergent results under identical input conditions, highlighting a lack of consistency. Consistency indicates that for the same input (e.g., one function), all models in the study give the same prediction (e.g., for a specific function, all methods predict it as defective or all agree that it is non-defective). Consistency can bring trust in prediction when developers make decisions. To assess the consistency of different models in defect detection, we also conducted an empirical study, as shown in Table I. Specifically, we fine-tuned two learning-based detection models (i.e., LineVul [9] and SVulD [28]) and two pre-trained models (i.e., UniXcoder [29] and CodeBERT [30]) respectively on three widely used defect detection datasets [23]–[25], and compute the consistency in different models. We can see that the consistency of all models is only 51.0% to 58.5%. Such low consistency will erode developers’ trust in the defect detection models, which hinders their practical application.

TABLE I: Consistency in different defect detection models

Models	Bears+Bugs.jar	Defects4J
Learning-based models	71.7%	65.0%
Pre-trained models	81.3%	79.5%
All models	58.5%	51.0%

B. Limited efficiency in unit test generation approaches

Unit test generation approaches have gained widespread attention and application due to their ability to directly identify defects in software. Existing unit test generation approaches can be categorized into two types: traditional generation approaches and learning-based generation approaches. However, both of them suffer from inefficiency issues.

Traditional generation approaches [31], [32] focus on the code coverage metric. Researches show that traditional generation approaches are very effective at achieving high coverage [33]–[36], even covering more code than manually written test cases. However, previous studies indicate that high code coverage does not always result in effective error triggering [17], [18].

Meanwhile, learning-based generation approaches [19]–[21] have been proposed and have made significant progress. These approaches are trained on specific test generation datasets (e.g., Methods2Test [22]), enabling them to generate test cases that meet specific testing objectives. They focus on randomly generating a large number of test cases, lacking effective information guidance. This leads to poor test efficiency, failing to efficiently trigger errors. Moreover, a dataset comprising high-quality pairings of defective methods with error-triggering test cases is essential for training a model that efficiently generates error-triggering test cases. However, existing approaches are typically trained on the Methods2Test [22] dataset, which contains numerous clean methods and test cases incapable of triggering errors. Consequently, a significant portion of the test cases generated by these approaches fail to trigger errors, resulting in limited efficiency in error triggering.

Intuition. *Providing corresponding error-triggering unit tests along with test results will help to instill greater confidence in defect detection results. Simultaneously, defect detection information can be leveraged to guide the generation of unit tests, reducing the model’s search space and enhancing the efficiency of unit test generation.*

III. APPROACH

We propose a WYSIWYG approach: **A**ttention-based **S**elf-guided **A**utomatic **U**nit **T**est **G**en**E**ration (AUGER), which contains two stages: defect detection and error triggering. In the former stage, AUGER first detects the proneness of defects. In the latter stage, it guides the LLM to generate unit tests for triggering such an error with the help of critical information present in defective code. As a result, AUGER will provide developers with defect detection results, convincing error-triggering unit tests, and corresponding test results. Although AUGER is general, in this paper, we adopt recent DeepSeek Coder [37] and CodeLlama [38] as the backend LLMs, which can be easily replaced with various state-of-the-art models (e.g., CodeT5+ [39] and StarCoder [40]). AUGER includes defect detection, attention-guided unit test generation, and unit test validation. Fig. 1 provides an overview of our approach:

- **Defect Detection.** We first introduce a Java class file and extract all methods in the class file using the JavaParser [41]

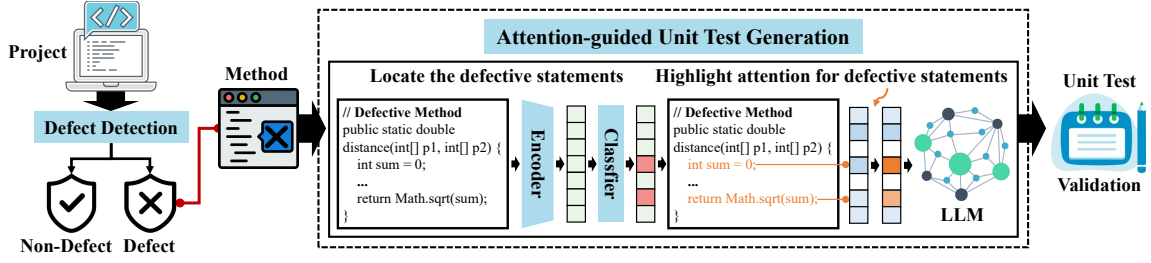


Fig. 1: Overview of AUGER

tool. We encode the methods into token representations and input them into AUGER to detect whether there are defects.

- **Attention-guided Unit Test Generation.** We conduct further analysis on the detected defective methods to identify the defective statements. Then, we guide the LLM to focus on the defective statements in order to generate unit tests that trigger the errors.
- **Unit Test Validation.** We describe how AUGER injects a unit test into an existing suite (i.e., test class) and elaborate on how AUGER adds the required dependencies to successfully execute the injected unit test.

A. Defect Detection

To improve robustness and performance, we adopt adversarial learning and contrastive learning framework with the pre-trained model, UniXcoder [29]. There are three important components of AUGER: (1) an encoder for embedding methods' semantics, (2) an attack strategy for generating adversarial samples, and (3) a learning strategy for discriminating differences between normal and adversarial samples.

1) *Code Encoder:* UniXcoder, proposed by Guo et al. [29], is a unified pre-trained model incorporating semantic and syntax information from both code comment and AST, and we adopt it as the code encoder in AUGER to embed the code features at the method level. UniXcoder transforms the input method to a 768-dimensional embedding E_M [42], [43].

2) *Adversarial Learning:* In adversarial learning, the goal is to enhance the robustness of the model against adversarial attacks. We employ FGM [44] to conduct adversarial learning on UniXcoder, producing the encoded samples after the attack, denoted as E_A . The process involves: (1) Perturbations are applied to the original input data to generate adversarial samples. These perturbations are crafted to challenge the model's ability to correctly classify or handle the input. (2) *Training with Adversarial Samples:* The model is trained on both the original and adversarial samples. This training helps the model learn to recognize adversarial inputs.

3) *Contrastive Learning:* Contrastive learning [28], [45], [46] is employed to enhance the model's resistance to perturbations by aligning normal and adversarial samples. The training objective is to fine-tune the network such that the encoded representations E_M (i.e., normal samples) and E_A (i.e., adversarial samples) are as close as possible. The objective can be described as $\max(\|E_M - E_A\|, 0)$, which encourages the network to reduce the distance between these representations.

We employ the KL-divergence loss used in R-Drop [45] to quantify the differences between normal and attacked samples

to minimize the distance between E_M and E_A . During the fine-tuning phase, we use the cross-entropy loss to guide the optimization process of AUGER by comparing the difference between the prediction probability of the model and the label. The final loss of defect detection consists of both classification cross-entropy loss and KL-divergence loss, which can be described by the following equation:

$$\mathcal{L}_{ce} = - \sum_i y_i \log(\hat{y}_i) \quad (1)$$

$$\mathcal{L} = \mathcal{L}_{ce} + \beta \cdot \mathcal{L}_{kl} \quad (2)$$

B. Attention-guided Unit Test Generation

LLM is pre-trained using millions of code snippets from open-source projects, showing dominantly superior reasoning capabilities over existing AI models in downstream tasks [47]–[51]. In this section, we aim to stimulate the powerful capabilities of LLM to efficiently generate error-triggering unit tests. To achieve this, we need to address four tasks: (1) **Defect Location**, (2) **Prompt Preparation**, (3) **Attention Profiling**, and (4) **Attention Inference**.

1) *Task 1: Defect Location:* Similar to the defect detection process, we also utilize UniXcoder as the foundational model for defect location, adopting it to embed code features at the statement level. More precisely, given the source code of a defective method, AUGER first splits the method into individual statements. Then, AUGER transforms the input method to $n \times 768$ -dimensional vectors at the statement level, where n indicates the number of statements. Finally, after passing the final representation through a fully connected layer and the *softmax* layer, AUGER outputs defective statements. The cross-entropy loss function is used to train the process.

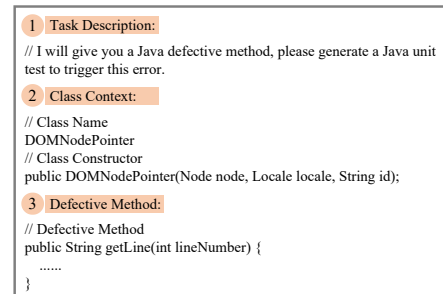


Fig. 2: An example of prompt for LLM

2) *Task 2: Prompt Preparation:* AUGER is generalizable and can be extended to other programming languages by modifying the language-specific information in the prompt (e.g.,

“Java” to “Python” and “/” to “#”). The prompt involves three important components as illustrated in Fig. 2:

- **Task Description** (marked as ①). LLM is provided with the description constructed as “// I will give you a Java defective method, please generate a Java unit test to trigger this error”.
- **Class Context** (marked as ②). The class name and class constructor provide key information about the structure and initialization process of the class.
- **Defective Method** (marked as ③). We provide the defective method in the method-level error-triggering scenario. We also prefix the defective method with “// Defective Method” to directly indicate LLM about the context of the method.

Due to the vast search space of LLMs, these models randomly generate entirely different unit tests for focal methods. To efficiently obtain high-quality outputs within a reasonable time frame, we have modified the attention mechanism of the LLM. This modification is designed to guide the model’s attention specifically toward the statements where defects are located. The specific process (Algorithm 1) consists of two components: (1) **Attention Profiling**, which selects the effective attention heads for modifying, and (2) **Attention Inference**, which emphasizes the defect location information of the defective methods during inference.

Algorithm 1: Attention Profiling and Attention Inference

Attention Profiling (Section III-B3)

Input: Profiling set \mathcal{D} , coefficient α , attention layer number L , attention head number H , hyperparameter k ;

for $l \leftarrow 1$ **to** L , $h \leftarrow 1$ **to** H **do**

- 1: Modify the attention head (l, h) by Equation 3;
- 2: Evaluate the unit test generation performance on \mathcal{D} ;
- 3: Collect the top k heads \mathcal{H} with the validation results;

Output: The attention head set \mathcal{H} ;

Attention Inference (Section III-B4)

Input: Prompts \mathcal{P} , defective statements \mathcal{S} , coefficient α ;

for head (l, h) in \mathcal{H} **do**

- 1: Modify the attention head (l, h) by Equation 3;
- 2: Generate a large number of candidate unit tests;

Output: Unit tests \mathcal{T} ;

3) *Task 3: Attention Profiling*: LLM typically has multiple attention layers and multiple attention heads (e.g., DeepSeek Coder 6.7B has 32 attention layers, each with 32 attention heads). Attention heads [52] are components of the multi-head attention mechanism in Transformer models. Each attention head operates independently, enabling the model to focus on different parts of the input sequence simultaneously. However, there is currently no consensus on the specific roles that these attention heads play. Zeng et al. [53] demonstrated that the first attention layer provides insight into which tokens the model focuses on. On the other hand, Wan et al. [54] showed that deeper attention layers excel in capturing long-distance dependencies and program structure. It is important to specify the correct attention heads, given that different heads serve distinctive roles in encoding semantic/syntactic information. To this end, we propose an attention profiling algorithm to identify

the effective attention heads for inference. Specifically, we sub-sample profiling set \mathcal{D} (100 samples) from the Defects4J dataset (cf. Section IV-A). After that, we need to define how to modify the LLM’s attention so that it focuses on the specified statement. AUGER emphasizes the defective statements of the input method by down-weighting the attention scores of tokens that are not in the defective statements. Specifically, given the tokens of highlighted statements as \mathcal{S} , AUGER emphasizes these tokens by an attention projection \mathcal{W} :

$$\mathbf{H}^{(l,h)} = \mathcal{W} \left(\mathbf{A}^{(l,h)} \right) \mathbf{V}, \text{ where } [\mathcal{W}(\mathbf{A})]_t = \begin{cases} \mathbf{A}_t / C & \text{if } t \in \mathcal{S} \\ \alpha \mathbf{A}_t / C & \text{else} \end{cases} \quad (3)$$

where $0 \leq \alpha < 1$ is a scaling coefficient and $\mathbf{A}^{(l,h)}$ denotes the attention scores at the head h of the l -th layer. The term $C = \sum_{t \in \mathcal{S}} \mathbf{A}_t + \sum_{t \notin \mathcal{S}} \alpha \mathbf{A}_t$ normalizes the scores so that they sum up to one. Attention modifying is conducted during the inference process and does not require any training.

Equation 3 modifies the model attention by scaling down the scores of tokens that are not in defective statements. When the coefficient α is set very small, defective statements are highlighted given their increased attention scores after re-normalization. As shown in Fig. 3, AUGER recognizes that line 6 is a defective statement. Consequently, AUGER first marks the defective line in the prompt, and then recalculates the attention weights within the attention heads, directing the LLM’s focus towards the tokens associated with the statement.

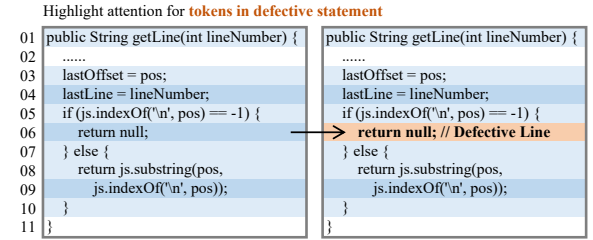


Fig. 3: Highlight attention for tokens in defective statement

After that, we assess the performance of modifying each individual attention head (l, h) , where $1 \leq l \leq L$ and $1 \leq h \leq H$, on a designated subset \mathcal{D} . We rank all the heads based on their unit test generation performance, specifically, by evaluating how many errors can be triggered on \mathcal{D} . Subsequently, we define the attention head set \mathcal{H} for inference as the top k performing heads.

Unlike fine-tuning, attention profiling doesn’t modify any model weights, so it demands similar computational resources as inference. The resulting head set \mathcal{H} serves as a model-level profile. Once determined, we can use attention inference on \mathcal{H} for both existing and unseen datasets, enhancing model comprehension and boosting performance.

4) *Task 4: Attention Inference*: During the inference process, AUGER modifies the attention head (l, h) in the attention head set \mathcal{H} using Equation 3. The input to this process includes the prompt (i.e., task description, class context, and defective method), defective statement, and the coefficient α , while the output consists of a large number of candidate unit tests.

C. Unit Test Validation

Our approach is automated and requires no manual intervention. Therefore, to validate whether the unit tests generated by LLM can accurately trigger identified errors, we need to automatically inject the unit tests into the corresponding test classes. Additionally, we must add the required dependencies to execute and obtain the test results.

1) *Inject unit test into test class*: We use token similarity to find the test classes that are most similar to the generated unit tests and inject them. The intuition is that, if a unit test belongs to a test class, the unit test likely uses similar methods and classes, and it shares similar tokens to other tests from that test class. Formally, we assign a matching score for each test class based on equation: $\text{sim}_{c_i} = |T_t \cap T_{c_i}| / |T_t|$, where T_t and T_{c_i} are the set of tokens in the generated unit test and the i -th test class, respectively.

2) *Add the required dependencies*: First, AUGER parses the generated unit test and identifies variable types and the referenced class names, constructors, and exceptions. AUGER then endeavors to locate public classes matching the identified type name for unimported dependencies. If precisely one such file exists, AUGER derives the classpath to the identified class and adds an import statement accordingly. However, there may be scenarios where no matching classes are found or multiple matches occur. In such cases, AUGER scans the project for *import* statements ending with the target class name and selects the most prevalent import statement across all files.

After injecting the unit test into the test class and adding the required dependencies, AUGER executes the test to check whether it triggers the identified errors.

IV. EXPERIMENT

In this section, we first present the studied datasets, and then introduce the baseline approaches. Following that, we describe the performance metrics as well as the experimental setting.

A. Datasets

1) *Defect Detection*: In order to ensure the thoroughness and validity of our research findings regarding defect detection, we have leveraged three widely used Java defect datasets: the **Bears** dataset [23], the **Bugs.jar** dataset [24], and the **Defects4J** dataset [25].

Since AUGER focuses on method level, we perform two filtering steps on the original datasets to obtain valid methods, and the filtering results of each dataset are displayed in Table II.

Step-1: Each commit is considered as a mini-version of a project. We use the commit IDs to request commit histories of the projects, and for each commit, we extract the code changes between before and after fixing a defect. Finally, we use the code change information to obtain the defective and fixed version of a method. Thus we collect the following information for a project: defective methods with their fixes and other clean methods. In this step, we obtain the Bears dataset, consisting of 2,009 methods, the Bugs.jar dataset, consisting of 40,880 methods, and the Defects4J dataset, containing 31,423 methods.

Step-2: To clean and normalize the dataset, we start by removing duplicate methods. The three datasets are derived from various versions of projects (e.g., Defects4J extracted from 17 real-world Java projects), leading to a substantial number of duplicates in methods extracted from different commits during step-1. In this step, we finally obtain the Bears dataset, which comprises 1,769 methods, the Bugs.jar dataset, which comprises 20,948 methods, and the Defects4J dataset, which comprises 17,806 methods.

2) *Unit Test Generation*: Defects4J includes utilities for generating and evaluating test suites on the programs to determine if generated tests pass on the fixed versions and catch defects on the defective versions. In contrast, Bears and Bugs.jar do not include readily executable test suites. Therefore, we evaluate real-world error triggering on the Defects4J dataset.

TABLE II: The statistic of studied datasets

Datasets	# Defective	# Clean	# Total	% Ratio
Bears	132	1,637	1,769	7.46%
Bugs.jar	1,953	18,995	20,948	9.32%
Defects4J	1,130	16,676	17,806	6.34%

B. Baselines

1) *Defect Detection*: To comprehensively compare the performance of existing work, we consider two learning-based detection approaches and two pre-trained models. The former group contains two approaches (i.e., LineVul [9] and SVuID [28]), which simply treat the source code as a sequence of tokens to represent its semantics. The latter group contains two models (i.e., CodeBERT [30] and UniXcoder [29]), which are pre-trained models for programming languages.

2) *Unit Test Generation*: We consider the following baselines in this evaluation:

Traditional Approach. We employ Randoop [32], a widely recognized tool extensively utilized for test case generation. Additionally, EvoSuite [31] is executed as a baseline, albeit its primary design for regression testing somewhat constrains its efficacy in triggering defects within the program. Both Randoop and EvoSuite are allocated a runtime of 3 minutes per defective class, adhering to the methodology outlined in [18], [21]. We then use scripts from previous works [21], [55] to run each test case and check if they trigger any errors.

Learning-based Approach. To present the learning-based approach, we employ the whole-test generation model, AthenaTest [20]. We also evaluate against TOGA [21], a unified transformer-based neural approach to infer both exceptional and assertion test oracles based on the context of the focal method. In addition, we fine-tune a seq2seq model, CodeT5+ [56], on the dataset used by TOGA to generate unit tests.

C. Evaluation Measures

1) *Defect Detection*: To evaluate the effectiveness of AUGER on defect detection, we consider the following metrics: Accuracy, Precision, Recall, F1-score, FPR, and PR-AUC.

Accuracy evaluates the performance that how many methods can be correctly labeled. It is calculated as: $\frac{TP+TN}{TP+FP+TN+FN}$.

Precision is the fraction of true defects among the detected ones. It is defined as: $\frac{TP}{TP+FP}$.

Recall measures how many defects can be correctly detected. It is defined as: $\frac{TP}{TP+FN}$.

F1-score is a harmonic mean of *Precision* and *Recall* and can be calculated as: $\frac{2 \times P \times R}{P+R}$.

FPR refers to the proportion of non-defects that are predicted to be defects. It is defined as $\frac{FP}{FP+TN}$.

PR-AUC is the area under the precision-recall curve and is a useful metric of successful prediction when the class distribution is very imbalanced [11].

2) *Unit Test Generation*: To evaluate the effectiveness of AUGER in triggering errors present in the program, the generated unit tests are run on the defective version and the fixed version. We consider an error as triggered if a generated unit test fails on the defective version and passes on the fixed version. Since each fixed version is distinguished from the defective version by a minimal patch fixing the specific error, a test must fail due to the specific error if it only fails on the defective version. In addition to evaluating the number of errors triggered, we use four metrics defined in [21], and we summarize the meaning of these metrics in Table III.

Following prior work [55], we also use the Precision metric. When using error triggering tools, developers care more about the Precision, i.e., how many FPs they need to inspect to trigger an error, and usually have few interests in using a tool with a low Precision [57], [58].

TABLE III: The metrics of error triggering

Metrics	Defective	Fixed
True Positive	Fail	Pass
False Positive	Fail	Fail
True Negative	Pass	Pass
False Negative	Pass	Fail

D. Experimental Setting

We implement AUGER in Python with the help of PyTorch [59] framework. All experiments are conducted on an NVIDIA A800 80GB graphics card. As for defect detection, we utilize *unixcoder-base-nine* [29] from Hugging Face [60] as our basic model. We fine-tune AUGER on the studied datasets to obtain a set of suitable parameters. During the training phase, we use *Adam* with a batch size of 16 to optimize the parameters of AUGER. We also leverage *GELU* as the activation function. A dropout of 0.1 is used for dense layers before calculating the final probability. We set the maximum number of epochs in our experiment as 200 and adopt an early stop mechanism. The models (i.e., AUGER and baselines) with the best performance on the validation set are used for the evaluations. As for unit test generation, we develop the generation pipeline in Python, utilizing PyTorch [59] implementation of DeepSeek Coder 6.7B and CodeLlama 7B. We use the Hugging Face [60] to load the model weights and generate outputs. During the attention profiling, we set α to 0.01 and k to 10. For each defective method, we generate 100 candidate unit tests (i.e., the candidate number is 100, refer to Section V-C for more details) and test them in the test suite provided by Defects4J.

V. RESULTS

To investigate the feasibility of AUGER on defect detection and error triggering, our experiments focus on the following three research questions:

- **RQ-1 Comparable Study of Defect Detection.** *How well does AUGER perform on method-level defect detection?*
- **RQ-2 Comparable Study of Error Triggering.** *How well does AUGER perform on triggering the error?*
- **RQ-3 Sensitivity Analysis.** *How do different configurations affect the overall performance of AUGER?*

A. RQ-1 Effectiveness on Defect Detection

Objective. Benefiting from the powerful representation capability of deep neural networks, many DL-based detection approaches have been proposed [9], [28]. CodeBERT and UniXcoder are bimodal pre-trained models for programming languages and natural languages, demonstrating excellent performance across various software engineering tasks, such as code search and code generation [29], [30]. The experiments are conducted to investigate whether AUGER outperforms SOTA method-level detection approaches.

Experimental Design. We consider four SOTA baselines: LineVul [9], SVuID [28], CodeBERT [30], and UniXcoder [29]. We conduct two distinct experiments to evaluate the performance of AUGER. In the first experiment, we undertake the tasks of training, validating, and testing using the Bears and Bugs.jar datasets. The second experiment extends our evaluation by training and validating on Bears and Bugs.jar, but testing on the Defects4J dataset. This experiment aims to showcase the detection capabilities of AUGER in identifying unknown real-world defects. It is noteworthy that, according to our statistical analysis, there is no overlap in the data extracted from Bears and Bugs.jar with that obtained from Defects4J. Our methodology for constructing the training and validation data from Bears and Bugs.jar aligns with established practices in prior research [9], [28], [61]. Specifically, 80% of methods are treated as training data, 10% of methods are treated as validation data, and the left 10% of methods are treated as testing data.

Results. The evaluation results on Bears and Bugs.jar datasets are reported in Table IV and the best performances are highlighted in bold. According to the results, we find that AUGER outperforms all SOTA baseline methods on almost all performance measures except *Recall*. In particular, AUGER obtains 0.353, 0.272, and 0.252 in terms of F1-score, Precision, and PR-AUC, which improves baselines by 4.7% to 7.6%, 17.7% to 20.4%, and 0.4% to 20.6% in terms of F1-score, Precision, and PR-AUC, respectively.

TABLE IV: Defect detection results of AUGER compared against four baselines on Bears and Bugs.jar

Methods	F1-score	Recall	Precision	PR-AUC
LineVul	0.331	0.585	0.231	0.231
SVuID	0.337	0.646	0.228	0.231
CodeBERT	0.328	0.598	0.226	0.209
UniXcoder	0.334	0.617	0.229	0.251
AUGER	0.353	0.502	0.272	0.252
<i>Improve</i>	4.7% to 7.6%	–	17.7% to 20.4%	0.4% to 20.6%

To evaluate the defect detection performance of AUGER on unknown real-world Java projects, we train and validate AUGER and baselines on Bears and Bugs.jar datasets. Subsequently, we test them on the Defects4J dataset. The performance comparisons of AUGER and four SOTAs on the Defects4J dataset are presented in Table V. According to Table V, we find that all SOTAs have poor performance on Defects4J dataset, while AUGER outperforms all baselines on almost all performance measures. Specifically, AUGER obtains 0.276 of F1-score, 0.198 of Precision, and 0.208 of PR-AUC, which improves baselines by 11.3% to 35.3%, 20.0% to 40.4%, and 24.6% to 69.1%, respectively. The results indicate that AUGER has a better learning ability than the four baselines.

TABLE V: Defect detection results of AUGER compared against four baselines on Defects4J

Methods	F1-score	Recall	Precision	PR-AUC
LineVul	0.230	0.406	0.160	0.165
SVulD	0.248	0.504	0.164	0.167
CodeBERT	0.204	0.369	0.141	0.123
UniXcoder	0.242	0.457	0.165	0.167
AUGER	0.276	0.453	0.198	0.208
<i>Improve</i>	11.3% to 35.3%	-	20.0% to 40.4%	24.6% to 69.1%

In the field of defect detection, the FPR and Accuracy are crucial metrics for assessing detection performance. FPR measures the extent of false alarms in the system, indicating the proportion of non-defective samples incorrectly flagged as defective. On the other hand, Accuracy provides a comprehensive evaluation of the overall correctness of the system. Therefore, we also compare the performance between AUGER and four baselines in terms of FPR and Accuracy. According to the results in Table VI, we can observe that AUGER exhibits a reduction in FPR ranging from 14.5% to 28.7% compared to other baselines. Additionally, its Accuracy shows an improvement ranging from 2.7% to 5.3% compared to other baselines. Notably, AUGER achieves the highest Accuracy while maintaining the lowest FPR, highlighting its credibility and usability in detecting defects in real-world Java projects.

TABLE VI: The performance between AUGER and four baselines in terms of FPR and Accuracy

Methods	FPR	% Decrease	Accuracy	% Improve
LineVul	0.145	14.5%	0.827	2.7%
SVulD	0.174	28.7%	0.806	5.3%
CodeBERT	0.152	18.4%	0.818	3.8%
UniXcoder	0.157	21.0%	0.818	3.8%
AUGER	0.124	14.5% to 28.7%	0.849	2.7% to 5.3%

Answer to RQ-1: AUGER outperforms the SOTA baselines at method-level software defect detection. Specifically, it achieves notable improvements in F1-score, Precision, PR-AUC, and Accuracy, as well as a reduction in FPR.

B. RQ-2 Effectiveness on Error Triggering

Objective. In order to enhance LLM’s proficiency in generating unit tests that trigger errors, we devised an attention profiling approach. This involves modifying the attention weights of the LLM to guide its focus toward statements within defective

methods that may harbor defects. This strategic adjustment aims to facilitate the generation of more unit tests that effectively trigger errors. In this section, our objective is to investigate whether AUGER outperforms previous unit test generation approaches in terms of error triggering.

Experimental Design. To facilitate comparison, we employ DeepSeek Coder and CodeLlama, denoting them as AUGER and AUGER*, respectively. We consider five SOTA baselines: TOGA [21], EvoSuite [31], Randoop [32], AthenaTest [20], and CodeT5+ [56].

For TOGA, we adhere to the approach outlined in [55], employing EvoSuite to generate test prefixes on the defective version. Subsequently, TOGA generates the corresponding test oracles. For EvoSuite and Randoop, we employ the scripts provided by the Defects4J toolkit. EvoSuite is utilized in its regression mode, while Randoop is applied in both regression and error-revealing modes (i.e., Randoop_{reg} and Randoop_{rev}). For each focal method, we set up AUGER and baselines to repeat 100 times, generating 100 candidate unit tests. As AUGER requires defect location, we fine-tuned a UniXcoder model to locate the defective statements within the methods using all the defective methods from Bears and Bugs.jar.

We have considered three experimental scenarios. The first scenario involves conducting experiments using defective methods detected in RQ-1. The second scenario entails using all defective methods from Defects4J for experimentation. The third scenario aims to evaluate whether AUGER can trigger errors in real-world projects. We follow Defects4J and collect defect-fixing commits from high-quality open-source projects included in Defects4J. We use DeepSeek Coder for exploration, which collected pre-training data from GitHub before February 2023 [37]. To prevent data leakage, we only collect defect-fixing commits from March 2023 onwards and obtain 61 defects. Then we extract methods following the steps in Section IV-A and ultimately obtain 41 method-level defects, which are used as input for AUGER to verify its ability to trigger errors in real-world projects.

Results. The effectiveness of AUGER compared against five baselines are reported in Table VII. According to the results, we can obtain the following observations:

(1) For the defects detected in RQ-1, TOGA, EvoSuite, Randoop_{reg} , Randoop_{rev} , AUGER*, and AUGER are able to trigger 26, 13, 18, 17, 32, and 35 errors, while AthenaTest and CodeT5+ are unable to trigger any errors.

(2) Similar to (1), for the 723 method-level defects in Defects4J, TOGA, EvoSuite, Randoop_{reg} , Randoop_{rev} , AUGER*, and AUGER could trigger 61, 26, 39, 35, 78, and 84 errors, respectively, while AthenaTest and CodeT5+ are unable to trigger any errors. AthenaTest and CodeT5+ generate a large number of candidate unit tests that fail to compile, which makes them extremely ineffective at triggering errors.

(3) Both AUGER and AUGER* show excellent performance, and AUGER works better than AUGER*. In the next section, we will select the best AUGER to study. In comparison to the baselines, AUGER triggers 9-35 errors and 23-84 errors more, highlighting the effectiveness of defective information

in guiding the LLM. In addition, our AUGER also has the highest Precision, meaning that developers only need to inspect a minimum number of unit tests under the same conditions.

TABLE VII: The effectiveness of AUGER compared against baselines on detected defects and all defects

Methods	Detected Defects (420)			All Defects (723)		
	Trigger	# Improve	Precision	Trigger	# Improve	Precision
TOGA	26	9	0.4%	61	23	0.7%
EvoSuite	13	22	0.1%	26	58	0.2%
Randoop _{reg}	18	17	4.6%	39	45	5.6%
Randoop _{rev}	17	18	1.9%	35	49	3.7%
AthenaTest	0	35	-	0	84	-
CodeT5+	0	35	-	0	84	-
AUGER*	32	6-32	5.1%	78	17-78	7.9%
AUGER	35	9-35	5.3%	84	23-84	8.8%

To better demonstrate the effectiveness of AUGER, we present the error-triggering quantities for different projects in two scenarios, as shown in Table VIII. According to the results, we can observe that: (1) Compared to other projects, AUGER excels at triggering errors in the Chart, Cli, Compress, and Gson projects, achieving a Recall of over 0.15 exceeding 0.15 in these projects. (2) AUGER performs poorly in projects such as JacksonXml, JXPath, and Mockito, where it fails to trigger any errors. Upon investigation, we found that most of these defects are multi-hunk defects involving multiple methods. Triggering these types of defects can be challenging.

TABLE VIII: The Recall of AUGER for the projects on Defects4J

Detected Defects (420)			All Defects (723)		
Projects	Recall	Prop.	Projects	Recall	Prop.
Chart	0.273	3/11	Chart	0.240	6/25
Closure	0	0/79	Closure	0.030	4/134
Collections	-	0/0	Collections	1	1/1
Csv	0	0/8	Csv	0.200	3/15
JacksonCore	0.077	1/13	JacksonCore	0.080	2/25
JacksonXml	0	0/3	JacksonXml	0	0/5
JXPath	0	0/13	JXPath	0	0/21
Math	0.139	10/72	Math	0.220	22/100
Time	0	0/11	Time	0.087	2/23
Cli	0.300	6/20	Cli	0.162	6/37
Codec	0.111	1/9	Codec	0.250	4/16
Compress	0.231	6/26	Compress	0.196	9/46
Gson	0.200	2/10	Gson	0.188	3/16
JacksonDatabind	0	0/62	JacksonDatabind	0.031	3/96
Jsoup	0.051	2/39	Jsoup	0.178	13/73
Lang	0.121	4/33	Lang	0.107	6/56
Mockito	0	0/11	Mockito	0	0/34
Sum	0.083	35/420	Sum	0.116	84/723

We draw a Venn diagram to further illustrate the performance difference on error triggering. For a better presentation, we independently illustrate the Top-4 best baselines (i.e., TOGA, EvoSuite, Randoop_{reg}, and Randoop_{rev}) on the basis of the number of triggering errors and ignore the baselines (i.e., AthenaTest and CodeT5+) that cannot trigger error for easy reference. Fig. 4 shows the illustrated results and we can also obtain two observations: (1) Individual approaches have unique abilities to trigger specific errors that others

cannot, making their performance somewhat complementary. (2) Overall, AUGER has a more powerful ability than baselines since it can trigger the most number of unique errors (i.e., 50) that other baselines can hardly trigger.

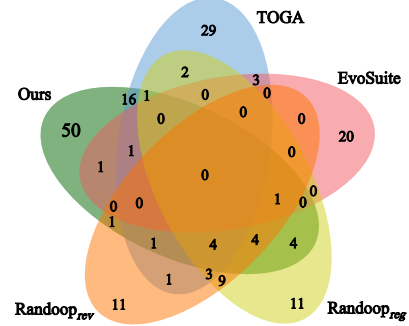


Fig. 4: Venn diagram of AUGER and studied baselines

Case Study. To explore why AUGER has an outstanding performance in triggering unique errors, we further analyze one example (i.e., Jsoup-85) as a case study, as shown in Fig. 5. The defect is that if key itself contains only space characters, then `key.trim()` will get an empty string. After the empty string is assigned to `this.key`, the `Validate.notEmpty(key)` check passes because key itself is not an empty string. But in reality, the value of `this.key` is an empty string, which is an incorrect state. Existing approaches cannot understand the defect present in the code, making it difficult to efficiently generate unit tests that trigger the error. As shown in Fig. 5, AUGER understands the semantics of the code correctly and generates a unit test that effectively exposes the defect in the defective method when handling key that only contain spaces. This example further exemplifies the capability of AUGER to leverage defective information to guide LLM to generate error-triggering unit tests efficiently.

Defective Method	Generated Unit Test
<pre>public Attribute(String key, String val, Attributes parent) { Validate.notNull(key); - this.key = key.trim(); + key = key.trim(); Validate.notEmpty(key); + this.key = key; this.val = val; this.parent = parent; }</pre>	<pre>public void test01() throws Throwable { Attributes atts = new Attributes(); String key = " ", val = "val"; try { Attribute att = new Attribute(key, val, atts); } catch (IllegalArgumentException e) { return; } fail("Expected an IllegalArgumentException"); }</pre>

Fig. 5: Unique error triggered by AUGER

Effectiveness of AUGER in real-world projects. Since the DeepSeek Coder’s pre-training data was sourced from GitHub prior to February 2023, in order to prevent data leakage, we collect defects from real-world projects starting from March 2023 to evaluate AUGER’s error-triggering capabilities. Table IX presents the results of AUGER for these real-world projects. According to the results, we can conclude that AUGER possesses the ability to trigger errors in real-world projects, not just limited to the defects present in the Defects4J dataset. This further demonstrates the practicality and feasibility of AUGER in generating unit tests. AUGER has the ability to guide LLM in generating effective unit tests, triggering potential errors in future real-world projects.

TABLE IX: The Recall of AUGER for the real-world projects

Project	Recall	Prop.	Project	Recall	Prop.
Cli	0	0/3	Jsoup	0.056	1/18
Codec	0.250	1/4	Lang	0.375	3/8
Compress	0.125	1/8	Sum	0.146	6/41

Answer to RQ-2: AUGER achieves better performance in both the defects detected in RQ-1 and all method-level defects in Defects4J, triggering 35 and 84 errors, respectively. AUGER can also trigger potential errors in future real-world projects.

C. RQ-3 Configurations of AUGER

Objective. In defect detection, we have incorporated *Adversarial Learning* and *Contrastive Learning* to enhance the performance and robustness of the UniXcoder. As a result, we aim to investigate the individual effects of these two components. In error triggering, we modify attention to guide LLM focus on defective statements, generating unit tests that trigger errors. During this process, we seek to examine the impact of attention modifying, as well as the influence of the candidate number of unit tests on the final results.

Experimental Design. First, we investigate the impact of different components on defect detection and design three variants of AUGER. AUGER-v1 represents the removal of *Adversarial Learning* and *Contrastive Learning*, AUGER-v2 indicates the removal of *Adversarial Learning*, and AUGER-v3 signifies the removal of *Contrastive Learning*. This approach allows us to examine the individual effects of each component. Subsequently, we explore the influence of attention modifying and defect location on error triggering, creating two variants. AUGER_{w/o} denotes the elimination of the attention-modifying component, while AUGER_{gt} denotes the utilization of Ground-Truth defect location to guide the LLM. Finally, we conduct a comparative analysis of the impact of candidate numbers on AUGER and TOGA, the latter of which is identified as the baseline with the best performance (cf. Section V-B).

Results. We discuss the results from the aspects of ablation and unit test candidate number, respectively.

TABLE X: Defect detection results of AUGER compared against variants

Methods	AL	CL	F1-score	PR-AUC	FPR
AUGER-v1	✗	✗	0.242	0.167	0.157
AUGER-v2	✗	✓	0.248	0.167	0.174
AUGER-v3	✓	✗	0.261	0.180	0.143
AUGER	✓	✓	0.276	0.208	0.124

Impact of components in defect detection. Table X shows the effectiveness of different variants and the better performance is highlighted in bold. According to the results, we can observe that: (1) Two components have their own advantages in a method-level defect detection scenario, achieving a varying performance and significantly improving the performance of AUGER-v1. Both of them can contribute

to the performance of AUGER. (2) *Adversarial Learning* demonstrates superior performance compared to *Contrastive Learning*. Specifically, AUGER-v3 outperforms AUGER-v2 in terms of F1-score (0.248→0.261), PR-AUC (0.167→0.180), and FPR (0.174→0.143) metrics. (3) A combination of these two components yields optimal performance in terms of F1-score (0.276), PR-AUC (0.208), and FPR (0.124). This suggests that incorporating *Adversarial Learning* and *Contrastive Learning* can enhance the effectiveness of the pre-trained model for defect detection.

TABLE XI: Error triggering results of AUGER compared against variants

Datasets	AUGER _{w/o}	AUGER	AUGER _{gt}
Detected Defects	28	35	44
All Defects	67	84	99

Impact of components in error triggering. According to the results in Table XI, we observe that: (1) AUGER and AUGER_{gt} trigger more errors than AUGER_{w/o}, which indicates the importance of the attention modifying. Particularly, AUGER and AUGER_{gt} can trigger 17 and 32 more errors than AUGER_{w/o} in all defects. (2) AUGER achieves comparable performance, obtaining similar conclusions in two different scenarios: weaker than AUGER_{gt} but stronger than AUGER_{w/o}. As expected, AUGER_{gt} performs the best, as it utilizes Ground-Truth defect location information to guide the LLM, avoiding the misguidance caused by errors in defect location.

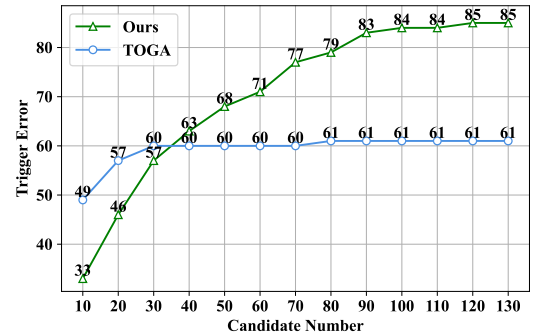


Fig. 6: The varying performance of AUGER and TOGA with different unit test candidate number on all defects

Impact of unit test candidate number. According to the results in Fig. 6, we find that: (1) Different candidate numbers have varying impacts on AUGER's performance and the performance of AUGER increases as the number of candidates increases. (2) Through the improvement curve, it can be observed that the performance improvement of AUGER far exceeds that of TOGA. As the number of candidate unit tests increases, the improvement curve of TOGA remains relatively flat, reaching optimal effectiveness when generating 80 candidate unit tests. In contrast, AUGER exhibits an upward trend, reaching optimal effectiveness when generating 120 candidate unit tests. (3) More candidate numbers may not guarantee additional performance improvement. When

AUGER generates 70 candidate unit tests, there is a significant improvement compared to generating 10 candidate unit tests (i.e., 33→77). However, when continuously increasing the number of candidates, the rate of performance improvement decreases (i.e., 77→85) and meanwhile, the generation cost with LLM is increasing. Considering both the performance improvement and the generation cost caused by LLM, we adopt 100 unit test candidate numbers as the default setting.

Answer to RQ-3: (1) The two components (i.e., Adversarial Learning and Contrastive Learning) contribute substantially to AUGER, and combining them achieves the best performance of defect detection. (2) Attention modifying can direct LLM to focus on defective statements to generate more unit tests that trigger errors. (3) Increasing the candidate number can significantly improve the error-triggering performance of AUGER, but will gradually become saturated.

VI. DISCUSSION

A. Comparison of Efficiency

Defect Detection. Table XII presents the details of time cost and GPU memory cost for baselines and AUGER. We find that although AUGER requires more time and computational resources during fine-tuning, its inference costs are comparable to those of the baselines. On the whole, during fine-tuning, AUGER takes an average of 3 minutes and 52 seconds per epoch and consumes 24,538M of GPU memory. In contrast, the baselines take an average of 1 minute and 42 seconds to 3 minutes and 24 seconds per epoch, with a GPU memory cost ranging from 13,412M to 23,442M. However, AUGER only requires 16 seconds and 5,658M of GPU memory during inference, which is very close to the baselines (i.e., 16 seconds for time cost and 4,626M to 5,754M for GPU memory cost). Given the performance improvements of AUGER (refer to Section V-A), the additional resources spent on fine-tuning are justified. Furthermore, AUGER does not require additional resources for inference after fine-tuning.

TABLE XII: The time cost and GPU memory cost of baselines and AUGER for one epoch on defect detection

Methods	Time Cost		GPU Memory Cost	
	Fine-Tuning	Inference	Fine-Tuning	Inference
LineVul	1m 16s	16s	13,924M	4,626M
SVulID	3m 24s	16s	23,442M	5,754M
CodeBERT	1m 42s	16s	13,412M	5,650M
UniXcoder	1m 45s	16s	13,416M	5,658M
AUGER	3m 52s	16s	24,538M	5,658M

Unit Test Generation. Fig. 7 shows the average runtime used by each baseline and AUGER on unit test generation. For TOGA, AthenaTest, and CodeT5+, we record the total fine-tuning time (i.e., FT in Fig. 7) and inference time during our experiments. TOGA uses the default settings from previous work [55], while AthenaTest and CodeT5+ undergo 10 epochs of fine-tuning. For EvoSuite, Randoop_{reg}, and Randoop_{rev}, which do not require fine-tuning and instead directly use scripts

provided by Defects4J to generate test cases, we only record the test case generation time (i.e., Inference in Fig. 7). Our AUGER does not require training, so we record the time for the attention profiling process (i.e., Profiling in Fig. 7) and the test case generation time. Obviously, we find that the total time required for AUGER is less than that of other baselines. Even though AUGER involves both profiling and inference processes, the time consumption for these processes is relatively low. For example, AUGER requires a total of 75.9 hours, whereas the baselines require between 78.3 hours and 156.7 hours. Although AthenaTest and CodeT5+ require minimal inference time, they require significantly longer fine-tuning times and ultimately perform poorly in error triggering (refer to Section V-B). Overall, AUGER uses the least total time and achieves the best error trigger results, making it more effective in real-world scenarios.

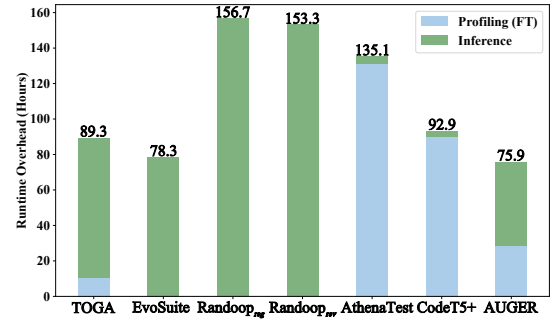


Fig. 7: The runtime overheads of baselines and AUGER

B. Threats to Validity

Internal Validity. The internal threat arises from potential data leakage since referenced unit tests may be part of the training data of LLM. To tackle this issue, we initially calculate the number of error-triggering unit tests generated by AUGER, which matches the reference unit test in Defects4J. We find that out of 84 triggered errors, 0 of error-triggering unit tests align with the unit tests in the fixed version. Only 12 unit tests are similar but not identical (e.g., input and output are different) to the unit tests in the fixed version. Additionally, compared to the basic LLM (i.e., DeepSeek Coder), AUGER demonstrates a significant enhancement in performance, triggering 17 more errors. This demonstrates that the improved results achieved by AUGER are not merely a result of memorizing the training data. Moreover, the pre-training data for DeepSeek Coder was collected from GitHub before February 2023 [37]. To prevent data leakage, we also collected defects from real-world projects starting from March 2023 to evaluate AUGER’s error-triggering capabilities. According to the results, AUGER can trigger 6 out of 41 errors. Therefore, we can conclude that AUGER possesses the ability to reveal potential errors in future real-world projects. The second internal threat arises from the reliance of unit test generation on defect detection. In this experiment, we conducted software defect detection on widely used datasets (e.g., Defects4J), and our model achieved the lowest FPR (i.e., 0.124) and the highest F1-score (i.e., 0.276), thereby mitigating the internal threat to the experimental design.

External Validity. The effectiveness observed in AUGER’s performance may not be applicable across different datasets. We conduct evaluations not only on the widely-used Defects4J dataset but also on the collected dataset from real-world projects. This broader evaluation scope aims to showcase the generalizability of our approach.

VII. RELATED WORK

A. Defect Detection

Traditional work on defect detection has explored a broad spectrum of metrics, encompassing factors such as code size [62], code complexity [63], object-oriented [62], organizational [64], and change history [65]), to forecast potential defects in software projects. Graves et al. [66] introduced the idea that recent modifications to code serve as effective indicators of impending defects, while Kim et al. [67] noted that defects often manifest in clusters within the history of software changes, meaning that recent changes and faults are likely to introduce defects in the future. DL-based approaches have been proposed to learn from historical data [1]–[4], [7], [9]–[13]. CodeBERT [30] and UniXcoder [29] are bimodal pre-trained models for programming languages and natural languages. They learn general representations that support downstream applications such as vulnerability detection [11], defect prediction [68], etc. Ni et al. [28] proposed SVulD by adopting contrastive learning to train the UniXcoder model for learning distinguishing semantic representation of functions regardless of their lexically similar information. However, these approaches often provide solely binary predictions, indicating whether a code snippet or statement contains defects or not [5]–[7]. To address this limitation, Fu et al. proposed LineVul [9], which uses attention scores to elucidate the reasoning behind the model’s decisions. Additionally, several approaches improve the model’s performance by integrating the graph structure of the code [10]–[15]. Just-In-Time (JIT) defect prediction approaches [69]–[71] have been proposed to predict whether a commit will introduce defects in the future. PyExplainer [72] is a novel, local, rule-based, model-agnostic technique designed to explain the predictions of JIT defect models. While these efforts have incorporated explanatory features, they often focus on why the model made a specific decision rather than providing detailed insights into the conditions that cause defects (e.g., the inputs and outputs that can expose defects).

Different from previous works, our paper connects defect detection and unit test generation. For each detected defect, AUGER generates corresponding test cases that trigger the error, instilling greater confidence in the defect detection results.

B. Unit Test Generation

Unit test generation approaches can be classified into two types: traditional generation approaches and learning-based generation approaches. However, both types face challenges related to inefficiency. Traditional generation approaches [31], [32] focus on code coverage, and research shows that traditional approaches are very effective at achieving high coverage [33]–[36]. Randoop [32] is a widely recognized tool extensively

utilized for generating unit tests for Java code using feedback-directed random test generation. EvoSuite is a tool that automates the generation of test suites, aiming for high code coverage, minimal size, and comprehensive assertions. However, previous studies show that high code coverage does not necessarily imply effective error triggering [17], [18]. Recently, learning-based generation approaches [19]–[21] have achieved significant progress. AthenaTest [20] is a Transformer-based model that is learned from developer-written test cases in order to generate correct and readable tests. The task is framed as a translation problem, where the source is a focal method and the target is the corresponding test case. TOGA [21] is a unified transformer-based neural approach to infer both exceptional and assertion test oracles based on the context of the focal method. A3Test [19] leverages the domain adaptation principles where the goal is to adapt the existing knowledge from an assertion generation task to the test case generation task. However, these approaches focus on randomly generating a large number of test cases by fitting the Method2Test dataset [22], which contains numerous clean methods and non-error-triggering test cases, lacking effective information guidance. This results in inefficient tests and an inability to efficiently trigger errors.

Different from existing works, our paper focuses on employing defect location information to guide the generation of unit tests, reducing the model’s search space and enhancing the efficiency of unit test generation.

VIII. CONCLUSION AND FUTURE WORK

We propose AUGER, which is a method-level approach for defect detection and error triggering. AUGER first detects the proneness of defects and then guides the LLM to generate unit tests for triggering such an error with the help of critical information present in defective code. As a result, AUGER will provide developers with defect detection results, convincing error-triggering unit tests, and corresponding test results. This automated approach instills confidence in developers regarding defect detection results. To evaluate the effectiveness of AUGER, we conduct a large-scale experiment by comparing it with SOTAs on the widely used dataset Defects4J. AUGER makes great improvements by 11.3% to 35.3%, 20.0% to 40.4%, and 24.6% to 69.1% in terms of F1-score, Precision, and PR-AUC, and can trigger 23 to 84 more bugs than SOTAs. Besides, we also conduct a further study to verify the generalization in practical usage by collecting a new dataset from real-world projects. In the future, we will generate multi-dimensional reports on the test results and conduct human studies to fully validate the effectiveness of AUGER.

ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China (Grant No.62202419), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), Zhejiang Provincial Natural Science Foundation of China (No. LY24F020008), the Ningbo Natural Science Foundation (No. 2022J184), and the State Street Zhejiang University Technology Center.

REFERENCES

- [1] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: Fault localization for deep neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 251–262.
- [2] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.
- [3] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.
- [4] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeeloctor: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.
- [5] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.
- [6] Q. Gao, S. Ma, S. Shao, Y. Sui, G. Zhao, L. Ma, X. Ma, F. Duan, X. Deng, S. Zhang *et al.*, "Cobot: static c/c++ bug detection in the presence of incomplete code," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 385–388.
- [7] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180.
- [8] X. Yin, C. Ni, and S. Wang, "Multitask-based evaluation of open-source llm on software vulnerability," *IEEE Transactions on Software Engineering*, 2024.
- [9] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.
- [10] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *In Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019, p. 10197–10207.
- [11] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th international conference on mining software repositories*, 2022, pp. 596–607.
- [12] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.
- [13] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.
- [14] 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.
- [15] Y. Li, S. Wang, and T. N. Nguyen, "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*, 2022, pp. 659–671.
- [16] B. Steenhoeck, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2237–2248.
- [17] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.
- [18] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.
- [19] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *arXiv preprint arXiv:2302.10352*, 2023.
- [20] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.
- [21] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2130–2141.
- [22] 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.
- [23] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An extensible java bug benchmark for automatic program repair studies," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 468–478.
- [24] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs. jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 10–13.
- [25] 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.
- [26] "Replication," 2024. [Online]. Available: <https://github.com/vinci-grape/AUGER>
- [27] C. Ni, L. Shen, X. Xu, X. Yin, and S. Wang, "Learning-based models for vulnerability detection: An extensive study," *arXiv preprint arXiv:2408.07526*, 2024.
- [28] C. Ni, X. Yin, K. Yang, D. Zhao, Z. Xing, and X. Xia, "Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1611–1622.
- [29] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.
- [30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.
- [31] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.
- [32] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.
- [33] A. Aleti, I. Moser, and L. Grunske, "Analysing the fitness landscape of search-based software testing problems," *Automated Software Engineering*, vol. 24, pp. 603–621, 2017.
- [34] C. Oliveira, A. Aleti, L. Grunske, and K. Smith-Miles, "Mapping the effectiveness of automated test suite generation techniques," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 771–785, 2018.
- [35] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 2015, pp. 1–10.
- [36] —, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.
- [37] D. AI, "Deepseek coder: Let the code write itself," <https://github.com/deepseek-ai/DeepSeek-Coder>, 2023.
- [38] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.
- [39] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.
- [40] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.
- [41] "Javaparser," 2024. [Online]. Available: <https://javaparser.org/>

- [42] W. Liu, S. Cheng, D. Zeng, and H. Qu, "Enhancing document-level event argument extraction with contextual clues and role relevance," *arXiv preprint arXiv:2310.05991*, 2023.
- [43] W. Liu, L. Zhou, D. Zeng, Y. Xiao, S. Cheng, C. Zhang, G. Lee, M. Zhang, and W. Chen, "Beyond single-event extraction: Towards efficient document-level multi-event argument extraction," *arXiv preprint arXiv:2405.01884*, 2024.
- [44] T. Miyato, A. M. Dai, and I. Goodfellow, "Adversarial training methods for semi-supervised text classification," *arXiv preprint arXiv:1605.07725*, 2016.
- [45] L. Wu, J. Li, Y. Wang, Q. Meng, T. Qin, W. Chen, M. Zhang, T.-Y. Liu *et al.*, "R-drop: Regularized dropout for neural networks," *Advances in Neural Information Processing Systems*, vol. 34, pp. 10 890–10 905, 2021.
- [46] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," *arXiv preprint arXiv:2104.08821*, 2021.
- [47] X. Yin, C. Ni, X. Xu, X. Li, and X. Yang, "Enhancing discriminative tasks by guiding the pre-trained language model with large language model's experience," *arXiv preprint arXiv:2408.08553*, 2024.
- [48] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung *et al.*, "A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity," *arXiv preprint arXiv:2302.04023*, 2023.
- [49] X. Yin, C. Ni, T. N. Nguyen, S. Wang, and X. Yang, "Rectifier: Code translation with corrector via llms," *arXiv preprint arXiv:2407.07472*, 2024.
- [50] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.
- [51] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.
- [52] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, E. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
- [53] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 39–51.
- [54] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture? a structural analysis of pre-trained language models for source code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2377–2388.
- [55] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards more realistic evaluation for neural test oracle generation," in *Proceedings of the 32th International Symposium on Software Testing and Analysis*. ACM, 2023.
- [56] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.
- [57] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.
- [58] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.
- [59] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems* 32. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [60] "Hugging face," 2023. [Online]. Available: <https://huggingface.co>
- [61] C. Ni, K. Yang, X. Xia, D. Lo, X. Chen, and X. Yang, "Defect identification, categorization, and repair: Better together," *arXiv preprint arXiv:2204.04856*, 2022.
- [62] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering*, vol. 33, no. 1, pp. 2–13, 2006.
- [63] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 9–9.
- [64] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 521–530.
- [65] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 284–292.
- [66] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.
- [67] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 489–498.
- [68] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: integrating semantic features with expert features for defect prediction and localization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 672–683.
- [69] Y. Zhao, K. Damevski, and H. Chen, "A systematic survey of just-in-time software defect prediction," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–35, 2023.
- [70] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes," *Empirical Software Engineering*, vol. 21, pp. 605–641, 2016.
- [71] C. Pornprasit and C. K. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 369–379.
- [72] C. Pornprasit, C. Tantithamthavorn, J. Jiarapakdee, M. Fu, and P. Thongtanunam, "Pyexplainer: Explaining the predictions of just-in-time defect models," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 407–418.