Cost-Effective Adversarial Attacks Against Code LLM With Model Attention

Weifeng Sun, Naiqi Huang, Meng Yan, *Member, IEEE*, Li Huang, *Graduate Student Member, IEEE*, Zhongxin Liu, *Member, IEEE*, Xiao Liu, and David Lo, *Fellow, IEEE*

***Abstract*—Code LLMs (CLLMs) are vulnerable to adversarial attacks, where semantically identical code mutations mislead models into incorrect predictions. To address this, adversarial training has been proposed, retraining models with adversarial examples generated by attack methods. Among various attack approaches, black-box methods have attracted increasing attention due to their flexibility and applicability. However, existing black-box attack methods face two key challenges: 1) vast mutation spaces limit attack efficiency and effectiveness, and 2) resource-intensive model queries constrain scalability. These challenges hinder the practicality of black-box attacks, especially under resource constraints, prompting the critical question: *Can we enhance the efficiency of existing attack methods without compromising their effectiveness?* To answer this, we conduct an empirical study using Explainable AI (XAI) techniques to investigate differences between adversarial and non-adversarial (failure) examples. After analyzing state-of-the-art attack methods against two CLLMs, we introduce the concept of *model attention deviation*, which quantifies differences in the model's focus between unmutated (original) and mutated code. Our findings reveal that adversarial examples exhibit significant attention deviations, with the direction of deviation critically affecting attack success. Building on these insights, we propose AdvSel, an efficient adversarial attack framework comprising two proxy components: the Attention Proxy Model (APM), which quickly estimates attention deviations to filter unpromising mutations, and the Deviation Direction Proxy Model (DDPM), which assesses whether attention shifts lead toward incorrect predictions. By integrating these proxy models with existing attack methods, AdvSel effectively prioritizes promising mutations, significantly improving attack efficiency. Experimental evaluations across five CLLMs, four downstream tasks, and three attack methods demonstrate that AdvSel maintains comparable attack success rates (a slight ASR reduction of 0.62%–0.70%) while significantly reducing model queries (by 34.98%–42.91%) and runtime (by 20.84%–21.45%). Under resource constraints, AdvSel consistently outperforms baselines, highlighting its practical advantage in cost-effective adversarial evaluation.**

***Index Terms*—Adversarial attack, code LLM, explainability.**

Received 22 June 2025; revised 7 January 2026; accepted 4 February 2026. Date of publication 16 February 2026; date of current version 20 April 2026. This work was supported in part by the National Natural Science Foundation of China under Grant 62372071, in part by the Fundamental Research Funds for the Central Universities under Grant 2022CDJDX-005, and in part by Chongqing Technology Innovation and Application Development Project under Grant CSTB2022TIAD-STX0007 and Grant CSTB2023TIAD-STX0025. Recommended for acceptance by S. Chattopadhyay. *(Weifeng Sun and Naiqi Huang contributed equally to this work.) (Corresponding author: Meng Yan.)*

Weifeng Sun, Naiqi Huang, Meng Yan, Li Huang, and Xiao Liu are with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China (e-mail: weifeng.sun@cqu.edu.cn; npcxh@cqu.edu.cn; mengy@cqu.edu.cn; lee.h@cqu.edu.cn; cdjx@cqu.edu.cn).

Zhongxin Liu is with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310027, China (e-mail: liu_zx@zju.edu.cn).

David Lo is with the School of Computing and Information Systems, Singapore Management University, Singapore 188065 (e-mail: davidlo@smu.edu.sg).

Digital Object Identifier 10.1109/TSE.2026.3663143

I. Introduction

CODE LLMs (CLLMs) [1], [2], particularly those built on advanced Transformers architectures [3], have achieved state-of-the-art (SOTA) performance across a range of code intelligence tasks, such as vulnerability prediction [4], [5]. These models are typically pre-trained on large-scale unlabeled datasets using self-supervised learning, followed by fine-tuning on labeled datasets for specific downstream tasks. The extensive programming knowledge gained during pre-training equips these models with strong semantic understanding and code generation capabilities.

However, like traditional models, CLLMs are susceptible to *adversarial attacks* [6], [7]. In the context of CLLMs, an adversarial attack occurs when the model generates significantly different outputs for two semantically identical input programs, where one is derived from the other through semantically preserving transformations [1], [8], [9], [10], [11], [12], [13], [14], [15]. As shown in Fig. 1, we refer to the input program before the application of transformations as the original example [Fig. 1(a) and (c)], and the CLLM under attack as the victim model. When a transformed version yields the same prediction as the original, it constitutes a failure example [Fig. 1(b)], indicating that the attack has not succeeded. In contrast, the modified/mutated code snippet that successfully misleads the victim model is referred to as an adversarial example [Fig. 1(d)]. This vulnerability is particularly concerning as CLLMs are increasingly deployed in mission-critical applications [12].

To address this vulnerability, *adversarial retraining* has been proposed, where adversarial examples are generated to identify weaknesses in the model and retrain it for improved robustness [12], [16]. Methods for automated adversarial example generation [1], [8], [9], [10], [11], [12], [13], [14], [15] are broadly categorized into white-box and black-box attacks. White-box attacks utilize internal model details, such as gradients, to craft effective adversarial examples. However, these methods often suffer from poor transferability across different models, as they rely on model-specific characteristics [15], [17], [18], [19]. Moreover, full access to the model's internal details may not

0098-5589 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

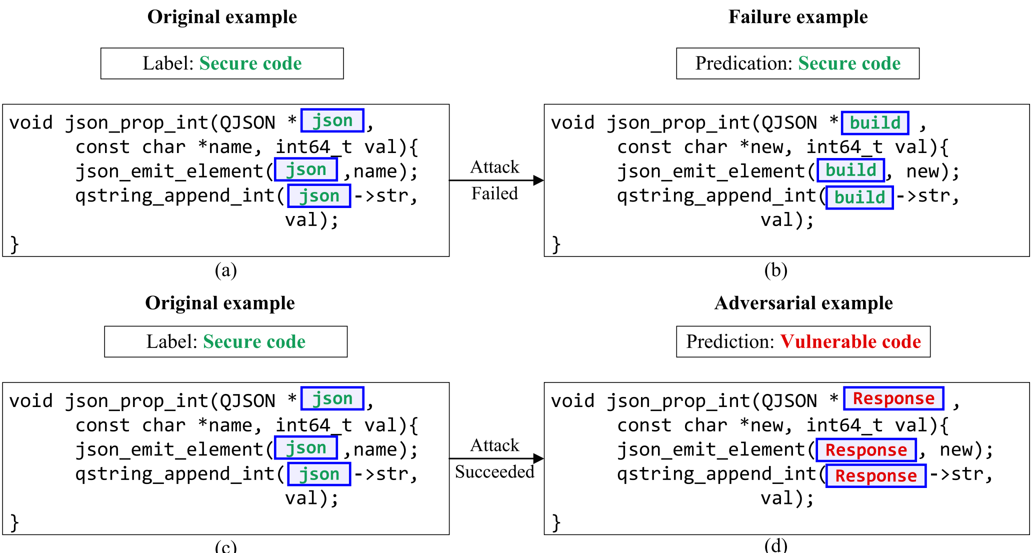


Fig. 1. Comparison among original examples, failure examples, and adversarial examples in code classification. Subfigures (a) and (c) show original examples. In (b), an adversarial rename produces a failure example where the classifier still predicts secure code, indicating the attack failed. In contrast, (d) demonstrates a successful adversarial example in which identifier substitutions cause the classifier to misclassify the same secure code as vulnerable code.

be available due to proprietary restrictions, further limiting the applicability of white-box methods [15]. In contrast, black-box attack methods treat the victim model as an opaque system, relying solely on its inputs and outputs to generate adversarial examples. Their flexibility and ease of implementation have driven increasing interest in these methods. Nevertheless, existing black-box attacks face two primary challenges, as outlined below.

Challenge 1: Vast mutation search space limits attack efficiency and effectiveness. With the many possible mutation locations and the extensive range of replaceable identifiers, the search space becomes overwhelmingly large. Existing methods often use greedy algorithms [12], [15] or restrict mutation types [15] to confine the search space to a more manageable size. While these approaches reduce computational overhead, they also compromise attack effectiveness by overlooking more diverse and potent adversarial examples, leaving certain model vulnerabilities undetected. Expanding the mutation space could yield stronger adversarial examples but at the cost of significantly higher resource demands.

Challenge 2: Resource-intensive model queries constrain attack scalability. Current black-box methods methods require querying the victim model for every mutated input. However, given the computational cost of model inference and the large number of mutations generated during the attack, this process is resource-intensive. With the rise and widespread use of large language models (LLMs), this problem becomes more pronounced. When testing resources are limited, generating adversarial examples becomes particularly challenging. This bottleneck significantly impairs the practicality of black-box attack methods in real-world applications.

Motivation. The abovementioned challenges prompt us to consider a key question: ***Can we improve the efficiency of existing attack methods without compromising their effectiveness—i.e., enhance their cost-effectiveness?*** In other words, when resources are limited, is it possible for testers to accurately identify adversarial examples from a large pool of mutations without querying the model for each mutation? Addressing this issue could offer significant benefits by reducing both the computational burden and the time required to interact with the victim model (*i.e.,* **Challenge 2**). Furthermore, reducing overhead would allow testers to expand a larger mutation space, increasing the likelihood of discovering adversarial examples (*i.e.,* **Challenge 1**).

Solution. Identifying adversarial examples from numerous mutations is akin to finding a needle in a haystack. To address this, it is crucial to *identify the fundamental differences between adversarial and failure examples*, where failure examples are mutations that do not cause incorrect predictions. Since these differences are hidden within the complex internal behaviors of CLLMs, we leverage explainable artificial intelligence (XAI) techniques to analyze model attention patterns. Specifically, we apply three SOTA black-box attack methods to two widely-used CLLMs: CodeBERT [20] and CodeT5 [21]. Given that CLLMs are more difficult to attack in understanding tasks than in generation tasks [15], we focus on vulnerability detection, code clone detection, and defect prediction. We employ four feature attribution methods from XAI: one self-attention-based and three gradient-based methods. These methods capture the ***model attention***, which indicates the importance the model assigns to different tokens. To quantify how adversarial and failure examples affect the model's behavior, we introduce the concept of ***model attention deviation***, which measures the difference in attention patterns between the original and mutated examples. Our evaluation yields four key findings: ① adversarial examples exhibit larger model attention deviations than failure examples; ② these deviations are detectable early in model layers and become more pronounced with depth; ③ attention deviation alone is not always a reliable indicator of adversarial success;

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

and ④ adversarial success critically depends on the directional alignment of attention deviations with attack objectives.

Gray-box Setting. Based on our empirical findings, we propose ADVSEL, a novel framework to enhance the cost-effectiveness of adversarial attacks. ADVSEL adopts a *gray-box attack setting*, where attackers/testers can access the model's output probabilities and shallow internal representations, such as early-layer attention weights. This assumption aligns with real-world APIs or local deployment environments, which often expose intermediate signals for interpretability or debugging [22], [23], [24]. Compared to white-box attacks requiring full model access, gray-box attacks offer a more practical balance between feasibility and effectiveness, enabling efficient filtering while retaining limited observability.

ADVSEL introduces two core components: the Attention Proxy Model (APM) and the Deviation Direction Proxy Model (DDPM). Drawing on Findings ① and ②, APM efficiently estimates attention deviations using only initial layers of the victim model, rapidly filtering mutations unlikely to become adversarial examples. Recognizing from Findings ③ and ④ that large attention deviations alone do not ensure adversarial success, DDPM employs a model probe [25] to evaluate whether the attention deviation directs the mutation toward incorrect classification. ADVSEL integrates seamlessly with existing attack techniques, generating a large pool of mutated samples and utilizing APM and DDPM to filter out less promising candidates. We evaluate ADVSEL across five CLLMs, four downstream tasks, and three attack methods. Results show that ADVSEL achieves performance similar to existing methods, with only a slight decrease in attack success rates (0.62% to 0.70%). Meanwhile, ADVSEL significantly improves efficiency by reducing model queries by 34.98% to 42.91% and runtime by 20.84% to 21.45%. When limiting model queries to 300, ADVSEL consistently outperforms baseline methods across all models and downstream tasks, achieving an average attack success rate improvement ranging from 4.71% to 55.09%, thereby demonstrating its strong effectiveness under resource constraints.

Novelty & Contributions. To sum up, the contributions of this paper are as follows:

(1) **Originality.** To our best knowledge, this work presents the first empirical study investigating the impact of adversarial examples on attention patterns in CLLMs across various code tasks.

(2) **Comprehensive Empirical Analysis.** We conduct an extensive investigation involving three adversarial attack methods, two CLLMs, and three downstream tasks. Our study explores critical aspects, including the distinguishing characteristics of adversarial examples, the reliability of attention deviations as indicators, and the underlying reasons why certain failure examples with large deviations do not result in incorrect predictions.

(3) **Improvement.** Based on our findings, we introduce ADVSEL, which streamlines the adversarial attack process by efficiently filtering out unlikely adversarial candidates. Our results demonstrate significant improvements in efficiency while maintaining attack effectiveness.

(4) **Open Science.** To support the open science community, we open-source all the studied datasets, experimental data, and our scripts for follow-up studies [26].

Paper Organization. Section II introduces background concepts and formalizes the problem. Section III outlines our empirical study design, including research questions, datasets, and analysis methods. Section IV presents the results of our empirical study, revealing key insights into the behavior of adversarial and failure examples. Section V details the design and implementation of ADVSEL, our proposed cost-effective adversarial attack framework. Section VI describes the experimental setup for evaluating ADVSEL. Section VII reports the experimental results, analyzing its performance across multiple models, tasks, and attack methods. Section VIII discusses the time complexity of ADVSEL and addresses potential threats to validity. Section IX reviews related work, and Section X concludes the paper.

II. BACKGROUND

A. Adversarial Example Generation

Our research focuses on understanding tasks, which are particularly challenging for adversarial attacks [15]. In this context, an adversarial attack involves applying slight mutations to a code fragment $x \in X$ to generate an adversarial example x_{adv} that misleads a classifier $f : X \rightarrow Y$ from its original prediction $y \in Y$. In black-box attacks, the attack process typically involves the following steps: 1) *Mutation Generation:* The tester/attacker generates syntactically correct and semantically equivalent mutations of the original code. 2) *Model Querying and Evaluation:* Each mutation is then inputted into the model to evaluate changes in prediction confidence compared to the original example, aiming to identify mutations that lead to incorrect predictions. 3) *Mutation Selection:* If none of the mutations deceive the model, the mutation that most significantly reduces the prediction confidence is selected as the seed for the next iteration. 4) *Iterative Refinement*: This process is repeated until an adversarial example is found or attack resources are exhausted.

B. Model Attention

Following prior work [3], [27], we define *model attention* as the importance a model assigns to different tokens during code understanding, indicating which parts of the input the model focuses on. This concept is closely related to *feature importance* [28] and *feature attribution* [29] in the XAI literature. Below, we describe the two types of attention computation methods employed in this study.

1) Self-Attention-Based Method: Transformer architectures employ the self-attention mechanism. The straightforward way to compute model attention involves using self-attention scores from Transformer-based models [30], [31], [32], [33]. However, aggregating these scores across different heads and layers is challenging, as each head may focus on different parts of the input. While various aggregation methods have been proposed, such as averaging across all layers [34] or emphasizing attention

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

from initial layers [1], there is no consensus on the optimal approach.

2) Gradient-Based Method: Gradient-based methods measure model attention by analyzing gradients of model predictions with respect to input tokens. Specifically, gradients computed through backpropagation indicate how strongly each token influences the model's output, with larger gradient magnitudes reflecting higher token importance.

III. Empirical Study Design

A. Research Questions

In this paper, we frame our empirical study around the following research questions:

RQ1: What differentiates adversarial examples from failure examples in CLLMs? We identify the key differences between adversarial examples that successfully deceive the model and failure examples—mutated inputs that do not lead to incorrect predictions.

RQ2: Can model attention deviation reliably indicate adversarial examples? Based on the insights from RQ1, we investigate whether significant deviations in model attention can reliably distinguish adversarial examples from failure examples.

RQ3: Why don't some failure examples with significant attention deviations become adversarial? Extending the investigation from RQ2, we explore why certain failure examples exhibit large attention deviations yet still fail to mislead the model.

B. Subjects and Datasets

1) Target Models: In this paper, we focus on code understanding tasks, which pose greater challenges for adversarial attacks [15]. Such tasks require models to accurately comprehend input code context, making encoder-only or encoder-decoder architectures preferable over decoder-only models. Specifically, for the encoder-only architecture, we choose CodeBERT [20]. For the encoder-decoder architecture, we select CodeT5 [21], which combines both encoding and decoding mechanisms, making it suitable for various tasks, including code understanding.

2) Adversarial Attack Approaches: We employ black-box attack methods to generate adversarial samples, focusing specifically on those that apply identifier-renaming transformations. Existing black-box attacks differ in how they use semantic-preserving transformations, operating either at the token level or statement level. Token-level methods, like identifier renaming, introduce smaller perturbations and are more stealthy. Statement-level transformations, such as adding dead code or converting `for` loops into `while` loops, tend to cause larger structural changes. To quantify this difference, we start from the same set of original programs and generate 300 successful adversarial examples using two types of transformations: identifier renaming and dead-code insertion. We then measure the edit distance between each adversarial example and its corresponding original program to characterize the extent of code modification. Our results show that identifier renaming results in substantially smaller changes (average edit distance = 0.0458), whereas dead-code insertion introduces much larger modifications (average edit distance = 0.3807). These findings empirically indicate that statement-level transformations move adversarial samples farther away from their original forms and reduce the stealthiness of the attack. Although such transformed samples remain semantically valid adversarial examples, they are less suitable for evaluating robustness under *minimal* and *stealthy* perturbations. Therefore, we restrict our study to three SOTA attack methods that use only identifier renaming: ALERT [12], WIR-Random [1], and RNNS [35].

- **ALERT [12].** ALERT utilizes the masked language prediction function of CLLMs (*e.g.,* CodeBERT) to identify and rank potential substitute tokens for each variable based on cosine similarity. ALERT operates in two phases: Greedy-Attack, which prioritizes impactful substitutions, and GA-Attack, which uses a genetic algorithm for a more thorough search if necessary. In line with the original settings, we set the number of candidate identifiers to 30 and the maximum number of iterations to $max\,(5 \times Num_i, 10)$, where Num_i denotes the number of identifiers in the code.
- **WIR-Random [1].** WIR-Random evaluates token importance by replacing them with a placeholder (*e.g.,* "UNK") and observing the impact on the model's predictions. Highly influential tokens are then randomly substituted with alternative identifiers, generating minimal yet effective perturbations. To maintain consistency, we limit the number of candidate identifiers to 30.
- **RNNS [35].** RNNS leverages historical attack data and continuous vector representations of variables to improve adversarial attacks. Initially, RNNS constructs a substitute set from real-world code datasets. It uses a pretrained variable name encoder to map these substitutes into a continuous vector space. These vector embeddings guide mutation selection, enhancing attack effectiveness. To avoid significant overhead in the attack process, the number of candidate identifiers is set to 30 in this study.

3) Downstream Tasks and Datasets: We investigate three downstream code understanding tasks: vulnerability detection, code clone detection, and defect prediction.

1) *Vulnerability Detection (VD).* Vulnerability detection involves identifying whether a given code snippet contains vulnerabilities [36], [37], [38]. We use the dataset prepared by Zhou et al. [39], extracted from two popular open-source C projects: FFmpeg [40] and Qemu [41]. This dataset consists of 27,318 functions labeled as either vulnerable or clean and is included in the CodeXGLUE benchmark [42]. We utilize the training, validation, and test sets as provided by CodeXGLUE.

2) *Code Clone Detection (CD).* Code clone detection aims to identify functionally identical or similar code segments. We use BigCloneBench dataset [43], a widely recognized benchmark containing over 6 million true clone pairs and 260,000 false clone pairs from Java projects. Following the ALERT [12],

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE I
STATISTICS OF USED SUBJECTS

Task	CLLM	Acc.	Number of Adversarial Examples		
			ALERT	WIR-Random	RNNS
Vulnerability Detection (VD)	CodeBERT	63.76%	761	926	1,135
	CodeT5	63.83%	916	1,160	664
Code Clone Detection (CD)	CodeBERT	96.97%	973	1,249	1,688
	CodeT5	98.08%	761	889	1,173
Defect Prediction (DP)	CodeBERT	86.69%	1,400	1,441	2,201
	CodeT5	88.54%	1,550	1,458	2,364

we use 90,102 examples for training and 4,000 examples for validation and testing.

3) *Defect Prediction (DP)*. Defect prediction is a program understanding task that aims to determine whether a given code snippet is defective and, if so, identify the type of defect. For this task, we use the CodeChef dataset [44], originally created by Zhang et al. [45]. The dataset comprises compilable C/C++ code snippets retrieved from the CodeChef platform[1], labeled based on the execution results from the platform: "OK" (no defect), "WA" (defect-1), "TL" (defect-2), and "RE" (defect-3).

C. Downstream Task Evaluation

We fine-tune the CLLM based on existing work [1], [42], and the replication results are listed in Table I.

D. Explainable Artificial Intelligence Methods

We examine four model attention computation methods: one based on self-attention and three based on gradients. Given a model $\mathcal{F}$ and an input sequence $X = \{x_1, x_2, \ldots, x_n\}$, we explain how each method calculates the model attention score c_i for each token x_i.

1) *Self-attention-based method*. To comprehensively analyze the model's focus, we compute attention scores for each layer. Since each layer contains multiple attention heads, we aggregate the scores across all heads by taking the maximum value, as described in [34]: $AggregatedAttn_l\,[i,j] = \max_{m=1}^{H} h_l^m\,[i,j]$, where $h_l^m\,[i,j]$ represents the attention that token x_i assigns to x_j in the m-th self-attention head of the l-th layer, and H is the number of heads. We then compute the model attention c_i for token x_i by summing the attention it receives from all other tokens [46]: $c_i = \sum_{j=1}^{n} AggregatedAttn_l[j,i]$.

2) *Gradient-based method*. We consider three gradient-based approaches to compute model attention: (1) *Saliency* [47] measures the impact of an input token on the model's output by calculating the absolute value of the gradient of the output with respect to the input token: $c_i = \frac{\partial \mathcal{F}(X)}{\partial x_i}$. (2) *Input × Gradient* [48] multiplies the embedding of the input token x_i by its gradient: $c_i = x_i \cdot \frac{\partial \mathcal{F}(X)}{\partial x_i}$. (3) *Integrated Gradients* [49] computes the average gradient along the path from a baseline input $\bar{X}$ to the actual input X. Let $\Delta x_i = x_i - \bar{x}_i$ and $\Delta X = X - \bar{X}$; the model attention is calculated as: $c_i = \Delta x_i \cdot \int_{\alpha=0}^{1} \frac{\partial \mathcal{F}(\bar{X}+\alpha\Delta X)}{\partial x_i} \mathrm{d}\alpha$.

[1]https://www.codechef.com/

E. Attention Alignment Calculation

Since CLLMs use different tokenization methods and adversarial attacks generally operate at the identifier level, we propose a method to map CLLM tokens back to the raw source code words and calculate attention alignment scores. First, we parse the code sequence into an Abstract Syntax Tree (AST) using `treesitter` [50], extracting the corresponding values from the terminal nodes, denoted as $\{t_{c,1}, t_{c,2}, \ldots, t_{c,N}\}$. Meanwhile, the model's tokenizer splits the code sequence into M tokens: $\{t_{m,1}, t_{m,2}, \ldots, t_{m,M}\}$. To compute the model's attention on the i-th code word $t_{c,i}$, we sum the attention scores of all CLLM tokens that overlap with $t_{c,i}$: $c_{c,i} = \sum_{\{t_{m,j} \cap t_{c,i} \neq \emptyset\}} c_{m,j}$. As shown in Fig. 2, the attention to the word `MessageDigest` ($t_{c,1}$) is the sum of the attention scores of the overlapping tokens: `Message` ($t_{m,1}$), `Dig` ($t_{m,2}$), and `est` ($t_{m,3}$).

F. Empirical Dataset Construction

We target the trained CLLMs from Section III-B1. Typically, attack methods do not generate adversarial examples for misclassified instances. To ensure a sufficient number of empirical samples, we select original examples from the training set, as victim models generally perform better on training data. Due to attack runtime constraints, we randomly sample instances to match the size of the test set. For each adversarial example, we retain a corresponding *failure example*, randomly selected from unsuccessful mutations in the final attack iteration. Our selection strategy is based on two main considerations: 1) *High Similarity*: These failure examples closely resemble adversarial examples, often differing by just one additional mutation, making them suitable for comparative analysis. 2) *Improved Generalizability*: By examining the differences between adversarial examples and their closely related failure counterparts, we can identify key distinctions that generalize across different attack stages. This methodology enhances the applicability of our empirical conclusions, enabling insights to extend not only to later-stage failure examples that closely resemble adversarial samples but also to early-stage failure examples with large token-level differences from adversarial examples. We report the number of generated adversarial examples in Table I.

IV. Empirical Results

A. RQ1: Adversarial vs. Failure Examples

1) Motivation: Improving the cost-effectiveness of adversarial attacks while maintaining their effectiveness is crucial. The main challenge is distinguishing adversarial examples from failure examples, given that the differences are often subtle and embedded within the complex mechanisms of CLLMs. Examining the model's attention patterns offers a promising solution, as previous research has shown that changes in model

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

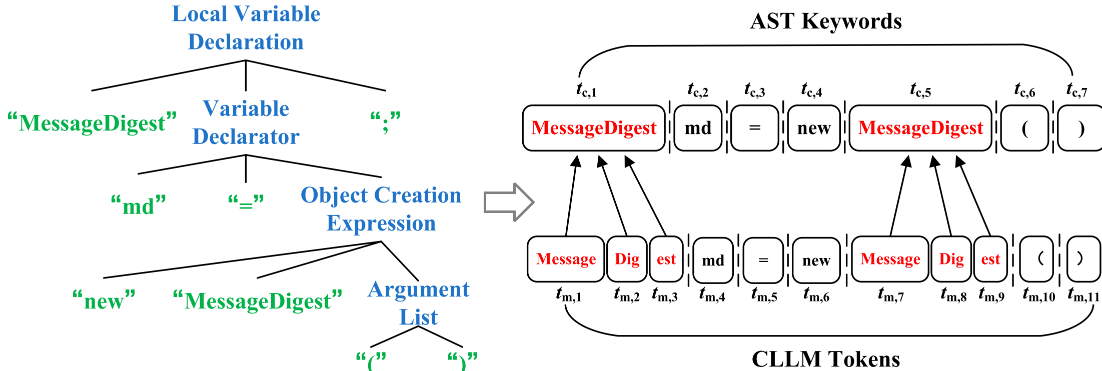


Fig. 2. Mapping CLLM sub-tokens back to code words.

attention can significantly impact the output of a model [30], [31]. By understanding how mutations affect the model's attention, we can gain valuable insights into the distinguishing characteristics of adversarial examples. We hypothesize that adversarial and failure examples cause different deviations in the model's attention patterns compared to the original example, termed *model attention deviation*. To validate this hypothesis, we employ explainable artificial intelligence (XAI) techniques to analyze the models' attention mechanisms.

2) Experimental Design: For each downstream task and CLLM, we apply three attack methods to generate adversarial examples and collect the corresponding failure examples (as detailed in Section III-F), along with the original examples. These sets are denoted as $ADV = \{adv_1, adv_2, \ldots, adv_m\}$, $Failure = \{f_1, f_2, \ldots, f_m\}$ and $Original = \{o_1, o_2, \ldots, o_m\}$, where m is the total number of examples generated by each attack method. For each sample o_i, adv_i, and f_i, we employ the four explainability methods from Section III-D: Self-attention, Saliency, InputX-Gradient, and Integrated-Gradient, to generate model attention scores. These scores reflect the importance of each input token to the model's prediction. Since these explainability methods operate at the token level, we apply the attention alignment method (Section III-E) to sum token-level attention into word-level attention scores. Different explainability methods can offer diverse insights and are often difficult to compare directly [51]. To standardize the results, we average the word-level attention scores obtained from Saliency, InputX-Gradient, and Integrated Gradients. For the Self-attention method, we extract attention scores for each layer to analyze the model's attention patterns at a fine-grained level. Next, for each o_i, adv_i, and f_i, we identify the top-k words with the highest attention scores (where $k = 5$), referred to as $M_{topKeyword}$. To quantify the model attention deviation, we calculate the overlap ratio of the top keywords between the mutated example and the original example:

$$AlignRate = \frac{|M_{topKeyword}(\mathit{Mutated}) \cap M_{topKeyword}(\mathit{Original})|}{|M_{topKeyword}(\mathit{Original})|} \quad (1)$$

The $AlignRate$ for each adversarial and failure example is denoted as $AlignRate_{adv}$ and $AlignRate_f$, respectively.

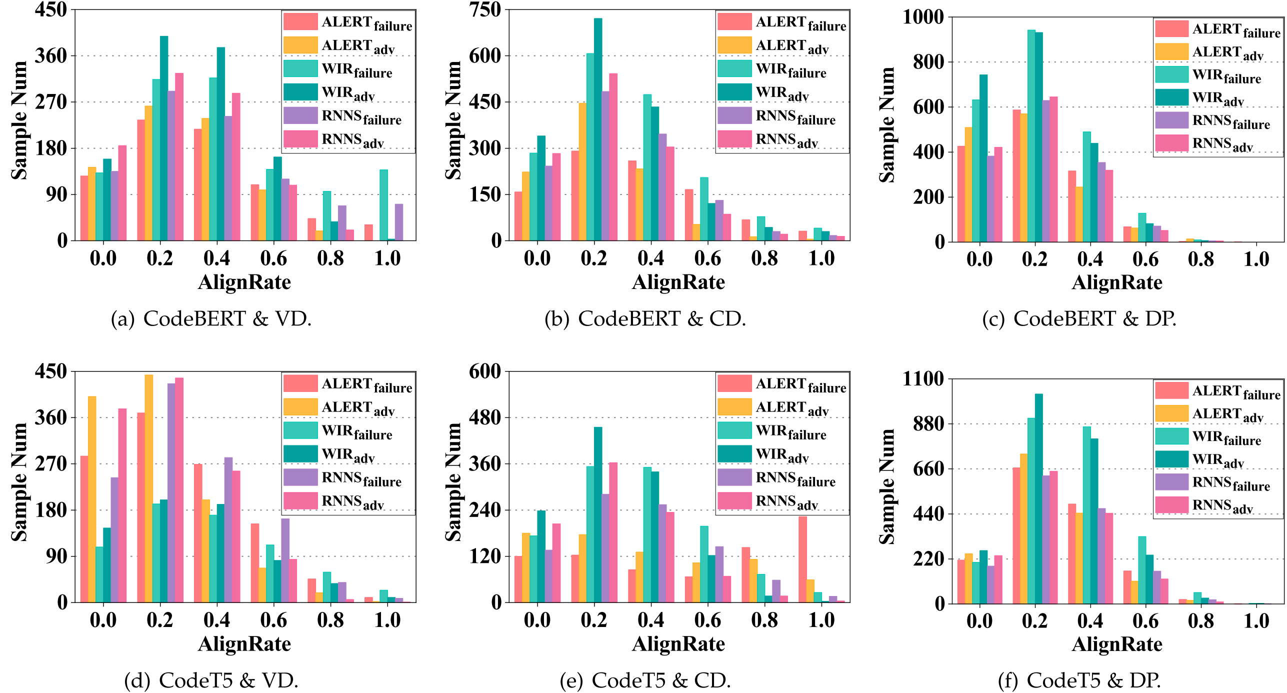
3) Results: Fig. 3 presents the $AlignRate$ for adversarial and failure examples across different models and tasks, highlighting the differences in attention alignment. The x-axis represents the $AlignRate$, and the y-axis shows the number of samples for each $AlignRate$ value. The following observation can be made: ① **Adversarial Examples Induce Higher Attention Deviation**: Across all models and tasks, adversarial examples show lower $AlignRate$ values. A Wilcoxon signed-rank test [52] confirms that this difference is statistically significant ($p < 0.005$).[2] The average $AlignRate$ for adversarial examples is 0.19–0.32 in CodeBERT and 0.20–0.39 in CodeT5, compared to 0.21–0.43 for failure examples in CodeBERT and 0.28–0.57 in CodeT5. For instance, in the CodeBERT&CD setting with the ALERT attack, when $AlignRate$ exceeds 0.8, there are 99 failure examples versus only 18 adversarial examples. Conversely, when $AlignRate$ falls below 0.2, adversarial examples dominate, with 669 instances compared to 449 failure examples.

> **Finding 1**: Adversarial examples exhibit significantly lower $AlignRate$ values than failure examples ($p < 0.005$), indicating more substantial deviations in model attention.

Fig. 4 presents the average $AlignRate_{adv}$ and $AlignRate_f$ across internal layers of CLLMs. We identify two key findings: ② **Early Detection of Alignment Differences**: The $AlignRate$ difference between adversarial and failure examples emerges in early layers across most models and tasks. Statistical analysis (Wilcoxon signed-rank test [52]) confirms significant differences in 15 of 18 experimental configurations (2 models × 3 tasks × 3 attack methods) from the first layer. Exceptions occur in three cases: ALERT for CodeBERT/CD (layer 6), WIR for CodeBERT/CD (layer 2), and RNNS for CodeBERT/CD (layer 5), where significance appears in deeper layers. These results demonstrate that adversarial mutations cause detectable disruptions in the model's internal representations during early processing stages. This finding shows the

[2]Due to space limitations, we have included all experimental results in our replication package [26].

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.



(a) CodeBERT & VD. (b) CodeBERT & CD. (c) CodeBERT & DP.

(d) CodeT5 & VD. (e) CodeT5 & CD. (f) CodeT5 & DP.

Fig. 3. **AlignRate** comparison using the gradient-based explainable analysis methods.

possibility of analyzing the model's internal layers to identify adversarial examples early, which could avoid complete model execution. ③ **Impact Amplified with Model Depth**: The $AlignRate$ divergence grows progressively across network layers, a trend consistently observed across most victim models and tasks. For instance, in CodeT5 on the VD task, the $AlignRate$ difference increases from 2.55% in the initial layer to over 24% in the final layer. This amplification occurs because attention deviations introduced by adversarial examples in early layers accumulate and intensify across successive layers. As a result, these growing deviations progressively distort the model's internal representations, ultimately leading to incorrect predictions.

> **Finding 2**: Attention deviations distinguishing adversarial and failure examples emerge early in model layers and amplify with increasing depth.

B. RQ2: Indicators of Adversarial Examples

1) Motivation: The findings from RQ1 demonstrate that adversarial examples induce significantly larger model attention deviations compared to failure examples. Building on this insight, RQ2 aims to determine whether such significant deviations reliably indicate adversarial examples.

2) Experimental Design: To evaluate the reliability of model attention deviations, we calculate the ratio of failure examples with a lower $AlignRate$ than their corresponding adversarial examples. A lower ratio indicates that attention deviations are more effective in distinguishing adversarial examples, while reducing the risk of misclassifying non-adversarial mutations.

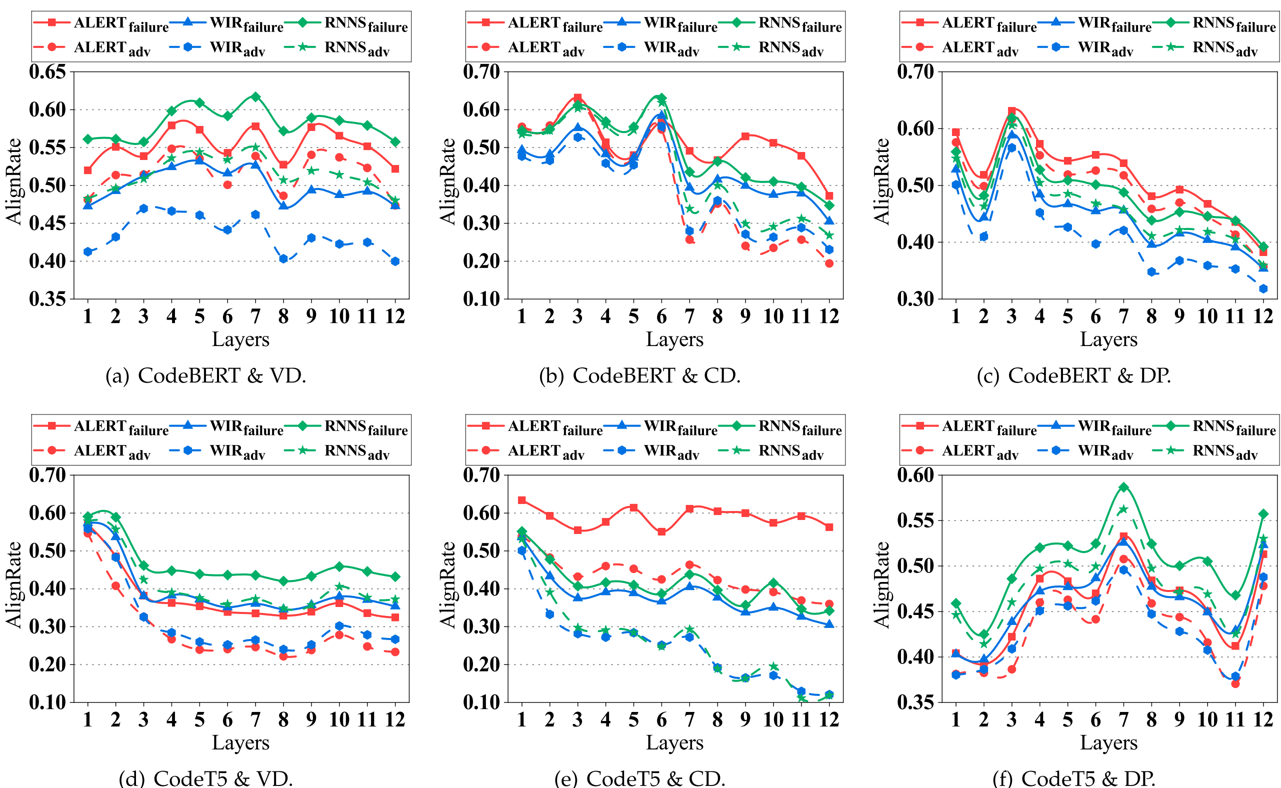
TABLE II
RATIO (%) OF FAILURE EXAMPLES WITH LOWER **ALIGNRATE**

CLLM	Method	VP	CD	DP
CodeBERT	ALERT	9.86	7.61	13.56
	RNNS	11.97	9.47	15.26
	WIR-Random	16.07	7.60	16.09
CodeT5	ALERT	12.19	5.91	13.86
	RNNS	15.94	7.67	13.95
	WIR-Random	13.18	7.30	16.11

We use the self-attention method due to its low computational cost. Specifically, we extract attention scores from the sixth layer of each CLLM for both adversarial and failure examples and calculate their $AlignRate$ relative to the original example. The sixth layer is selected as it marks the point where attention deviations begin to show notable differences across all tasks (as detailed in Section IV-A3).

3) Results: Table II presents the ratio of failure examples whose $AlignRate$ is lower than that of their adversarial counterparts. Overall, ALERT consistently yields the smallest ratios across the three tasks. Conversely, WIR-Random tends to exhibit the highest ratios—particularly for CodeBERT (16.07% in VP and 16.09% in DP) and CodeT5 (16.11% in DP)—implying that its attention deviations are less effective at distinguishing adversarial examples. RNNS also shows relatively higher ratios than ALERT, most notably on CodeBERT&CD (9.47%) and CodeT5&VP (15.94%), although it remains slightly more stable than WIR-Random in some cases.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.



(a) CodeBERT & VD. (b) CodeBERT & CD. (c) CodeBERT & DP.

(d) CodeT5 & VD. (e) CodeT5 & CD. (f) CodeT5 & DP.

Fig. 4. **AlignRate** comparison using the self-attention explainable analysis method.

> **Finding 3**: While model attention deviations can indicate adversarial examples, their effectiveness varies across CLLMs and attack methods. The proportion of failure examples exhibiting lower $AlignRate$ than their adversarial counterparts ranges from 5.91% to 16.11%, depending on the model and attack method. These results suggest that relying solely on attention deviations may introduce noise, as a non-negligible fraction of failure examples also present strong deviations.

C. RQ3: Manual Analysis

1) Motivation: Our empirical analysis in RQ2 reveals that some failure examples exhibit larger model attention deviations than adversarial examples, suggesting that attention deviation magnitude does not always directly correlate with prediction errors. This finding motivates us to investigate why such failure examples, despite exhibiting substantial attention deviations, fail to induce erroneous model predictions.

2) Experimental Design: We collect all failure examples where the model attention deviations exceed those of the corresponding adversarial examples, totaling 2,848 samples. From this set, we randomly sample 380 failure examples for manual analysis, ensuring a 95% confidence level and a 5% margin of error. Our analysis focuses on the relationship between their model attention deviations and changes in the model's prediction confidence.

3) Results: Our quantitative analysis reveals two distinct patterns in such failure examples: 1) **Unexpected Confidence Boost (6.84% of cases)**: A minority of failure examples (6.84%) unexpectedly exhibit increased prediction confidence for the correct label compared to their original example, despite substantial attention deviations. 2) **Limited Confidence Reduction (93.16% of cases)**: The majority of failure examples reduce model confidence, but not enough to alter the final prediction (mean confidence drop: 36.39% vs. 77.70% in successful adversarial examples). To analyze these patterns at the representation level, we first present two attention-based cases in Fig. 5(a) and (b). In Fig. 5(a), we analyze a seed input (S) and two mutations (M_1 and M_2). Although M_2 undergoes a smaller attention deviation than M_1, it induces a larger shift in the model's logits, effectively steering the model toward an incorrect prediction. This suggests that M_2 follows a more effective adversarial path due to the direction of its attention deviation. In contrast, Fig. 5(b) shows a failure case where mutation M_1 exhibits a large attention deviation, yet increases the model's confidence in the correct label. This example demonstrates that large attention deviations can sometimes reinforce, rather than misdirect, the model's decision process. We further substantiate these representation-level observations with concrete code-based examples in Tables III and IV. Specifically, Table III (Case 1) corresponds to the *Limited Confidence Reduction* pattern: although Mutation A introduces a relatively large attention deviation, the prediction remains correct, with confidence decreasing from 0.96 to 0.70. By contrast, Mutation B in the same table induces a smaller attention deviation but flips the prediction, consistent with the behavior observed in Fig. 5(a). Similarly, Table IV (Case 2) provides a code-level

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE III
CASE 1: ILLUSTRATING THE "LIMITED CONFIDENCE REDUCTION" PATTERN (CASE 1)

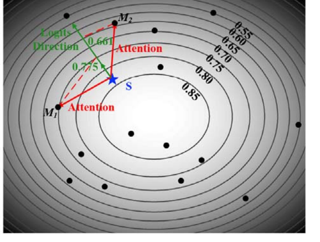
Mutation A: Failure example

```
  int main() {
      // ... (headers omitted)
      for (i = 1; i <= a, i <= b; i++) {
          if (a % i == 0 && b % i == 0) {
-             g = i;
+             rev_idx = i;
          }
      }
-     l = a * b / g;
-     printf("%d %d\n", g, l);
+     l = a * b / rev_idx;
+     printf("%d %d\n", rev_idx, l);
  }
```

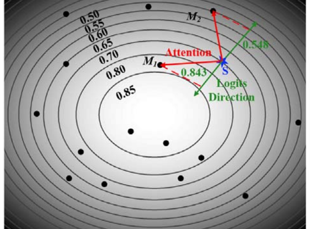
Mutation B: Adversarial example

```
  int main() {
      // ... (headers omitted)
      for (i = 1; i <= a, i <= b; i++) {
          if (a % i == 0 && b % i == 0) {
-             g = i;
+             findMin = i;
          }
      }
-     l = a * b / g;
-     printf("%d %d\n", g, l);
+     l = a * b / findMin;
+     printf("%d %d\n", findMin, l);
  }
```

	Mutation A: Failure example	Mutation B: Adversarial example
Attention Deviation:	1.11×10^{-4}	4.88×10^{-5} (Small)
Prediction:	**Correct** (Class 1)	**Wrong** (Class 3)
Confidence:	$0.96 \rightarrow 0.70$ (Dropped)	$0.96 \rightarrow 0.39$ (Flipped)



(a) Large attention deviation but small change in logits.



(b) Large attention deviation but opposite change in logits.

Fig. 5. Two illustrative examples.

instance of the *Unexpected Confidence Boost* pattern. Mutation A preserves the original prediction while increasing confidence ($0.63 \rightarrow 0.69$), despite noticeable attention deviation, whereas Mutation B successfully alters the model's decision. These results highlight that the *direction of attention deviation*, rather than its magnitude alone, determines whether a mutation leads to adversarial success or failure.

> **Finding 4**: Most failure examples (93.16%) exhibit ineffective directional deviations that reduce prediction confidence insufficiently. Additionally, some failure examples (7.37%) present counterproductive attention deviations that inadvertently reinforce correct features. This highlights that the attention deviation direction, rather than its magnitude alone, is the key factor in adversarial success, as effective attacks require alignment with adversarial objectives.

V. COST-EFFECTIVE ATTACK FRAMEWORK

Usage scenario. To improve the cost-effectiveness of existing attack methods, we propose ADVSEL, a novel framework that enhances attack efficiency while maintaining effectiveness. Designed as a plug-and-play solution, ADVSEL integrates seamlessly with existing attack techniques. It generates a large pool of mutated samples using current attack methods and then employs two key components—the **Attention Proxy Model** (APM) and the **Deviation Direction Proxy Model** (DDPM)—to identify suspicious samples more likely to mislead the model. By filtering out unlikely mutation candidates, ADVSEL significantly reduces the need for frequent queries to the victim model. Both APM and DDPM are informed by insights from our empirical study:

Insight 1: Our study shows that *model attention deviation* is a critical indicator for distinguishing adversarial examples from failure examples in CLLMs (**Finding 1**). Adversarial examples induce larger attention deviations than failure examples. However, computing model attention scores, especially via gradient-based methods, requires a full model pass, which is computationally expensive. Furthermore, we found that differences in attention deviations are observable in the model's internal layers and become more pronounced with increasing depth (**Finding 2**). This insight inspires the development of an attention proxy model that efficiently captures attention deviations based on the initial layers of the model without requiring full model passes.

Insight 2: Not all mutations that cause significant attention deviations are adversarial examples (**Finding 3**). Relying solely on attention deviations to identify adversarial examples would introduce noise. Our manual analysis indicates that the *attention deviation direction* is crucial (**Finding 4**); it determines whether a mutation moves the model toward incorrect classification. This insight motivates the design of a deviation direction proxy model to assess whether the direction of deviation supports generating effective adversarial examples.

A. Framework Design

1) Overview of AdvSel Workflow: The overall workflow of ADVSEL is depicted in Fig. 6, and its procedural steps are summarized in Algorithm 1. Given a seed input x, the framework first applies an existing attack method to generate a set of mutations $\mathcal{M}_t$. Rather than querying the victim model f

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
CASE 2: ILLUSTRATING THE "UNEXPECTED CONFIDENCE BOOST" PATTERN

Mutation A: Failure example

```
  // Context: int gcd(int a, int b) { ... }
  int main() {
      // ... (variable declarations)
      for (i = 0; i < n; i++) {
          scanf("%llu %llu", &n1, &n2);
          g = gcd(n1, n2);
          // Original used variable 'l'
-         l = (n1 * n2) / g;
-         printf("%d %llu\n", g, l);
          // Mutation: Variable 'no_of_queries'
+         no_of_queries = (n1 * n2) / g;
+         printf("%d %llu\n", g, no_of_queries);
      }
  }
```

Mutation B: Adversarial example

```
  // Context: int gcd(int a, int b) { ... }
  int main() {
      // ... (variable declarations)
      for (i = 0; i < n; i++) {
          scanf("%llu %llu", &n1, &n2);
          g = gcd(n1, n2);
          // Original used variable 'l'
-         l = (n1 * n2) / g;
-         printf("%d %llu\n", g, l);
          // Mutation: Variable 'deno'
+         deno = (n1 * n2) / g;
+         printf("%d %llu\n", g, deno);
      }
  }
```

	Mutation A: Failure example	Mutation B: Adversarial example
Attention Deviation:	4.21×10^{-5}	9.18×10^{-6} (**Very Small**)
Prediction:	**Correct** (Class 1)	**Wrong** (Class 3)
Confidence:	$0.63 \rightarrow$ **0.69** (**Boosted!**)	$0.63 \rightarrow 0.33$ (Flipped)

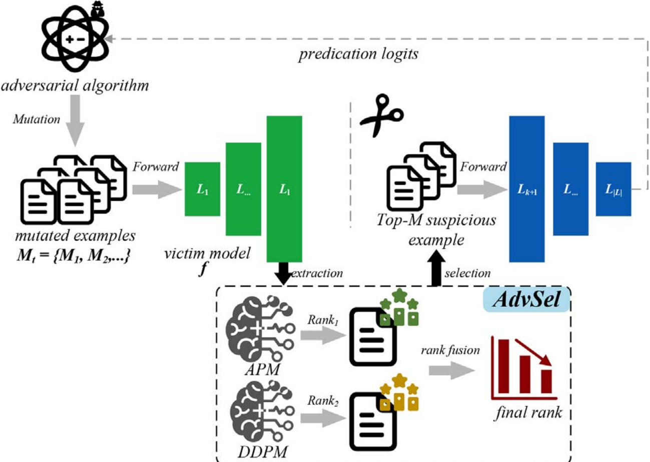


Fig. 6. Overview of ADVSEL method.

with all mutations, ADVSEL performs an efficient early-layer analysis: each mutated example is forwarded only through the first l layers of f, and the resulting intermediate representations are extracted for proxy evaluation. APM and DDPM then independently score these mutations from complementary perspectives, producing two ranked lists that reflect their estimated adversarial potential. A rank-fusion procedure integrates the two rankings into a unified priority ordering, from which the top-M suspicious mutations are selected. Only these selected candidates are forwarded through the remaining layers of the victim model to compute the final predication logits, thereby substantially reducing the number of full-model queries. Below, we detail each component of the framework.

2) Attention Proxy Model (APM): APM leverages internal layer information to efficiently capture attention deviations, as per **Findings 1** and **2**. Specifically, we select the l-th attention layer of the victim model f and decompose f into two parts: $f(x) = f^{(>l)}(f^{(\leq l)}(x))$, where $f^{(\leq l)}$ represents the first l layers and $f^{(>l)}$ represents the remaining layers. The first l layers are used to construct APM. For each sample m_j in $\mathcal{M}t$, APM captures its attention scores (**Line 4**) and computes the cosine similarity between these scores and those of x (**Line 5**): $\mathrm{CosSim}(x, c_j) = \frac{\mathbf{c} \cdot \mathbf{c_j}}{\|\mathbf{c}\| \|\mathbf{c_j}\|}$, where $\mathbf{c}$ and $\mathbf{c_j}$ are the attention scores vectors of x and m_j, respectively. This similarity calculation measures the degree of model attention deviation between the seed and mutated samples. Mutations with larger deviations (*i.e.,* lower cosine similarity) are ranked higher, as they are more likely to be adversarial.

3) Deviation Direction Proxy Model (DDPM): DDPM relies on model probes [25], a commonly used technique for interpreting deep neural networks, to assess the direction of attention deviation.

❶ Model Probe Design and Training. The probing network leverages internal CLLM representations to evaluate whether a mutation drives the model toward an incorrect prediction. Prior studies suggest that feed-forward network (FFN) modules serve as memory components of self-attention mechanisms, storing critical semantic information [53], [54]. To construct the DDPM, we use a simple yet effective two-layer linear classifier after empirical validation. Although this design may appear minimal, it is well-suited for the specific tasks in this study, minimizing the risk of overfitting while preserving the capacity to learn the necessary representations. The probing network takes the FFN hidden representations from the l-th layer (the same l as in APM) as inputs.

Definition 1: Let $f(\cdot; \theta)$ be a CLLM, and let x denote the input. A model probe p_l at the l-th layer of the CLLM is defined as: $p_l : f^l(x; \theta) \rightarrow [0, 1]^C$, where $f^l(x; \theta)$ is the output at layer l and $[0, 1]^C$ represents the categorical distribution over the C classes.

For each correctly predicted sample (x_i, y_i) from the training dataset, the representation h_i is extracted from the FFN hidden layer of the l-th layer. These representations can be directly obtained during the model's training phase, which is a common practice in various studies [30], [55]. The probing network p_l

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: AdvSel Pseudocode

```
Input: Victim model f, original example x, mutation
       set M, attack algorithm A, max iterations T,
       selected sample number M
Output: Adversarial example x_adv or failure example
        x_fail
1  Initialize APM, DDPM ; // Attention and
     Deviation Direction Proxy Models
2  for t ← 1 to T do
3      M_t ← A(x, M) ; // Generate mutated samples
         using existing adversarial attack algorithm
4      scores_apm ← APM(M_t, x) ; // Calculate
         attention scores
5      rank_apm ←
         argsort(−cosine(scores_apm, scores_apm(x))) ;
         // Rank based on cosine similarity with x
6      rank_ddpm ← DDPM(M_t, x) ; // Evaluate
         deviation direction ranking
7      rank_final ← RRFscore(rank_apm, rank_ddpm);
8      S ← {m_i | i ∈ top-M indices from rank_final} ;
         // Select top-M suspicious samples
9      for each m ∈ S do
10         Query victim model f with m;
11         if f(m) ≠ f(x) then
12             return x_adv = m ; // Successful
                 adversarial example
13         end
14     end
15     x ← arg min_{m∈S} f(m) ; // Update x with
         mutated sample minimizing f(x) on the
         ground truth
16 end
17 return x_fail = x ; // No adversarial example found
```

is then trained to predict the true label y_i by minimizing the cross-entropy loss:

$$\mathcal{L}(\theta) = -\sum_i y_i \log(p_l(h_i)) \tag{2}$$

❷ Evaluation. DDPM evaluates the attention deviation direction by first calculating the predicted probability of the true label for the seed input x: $p(y_i \mid x) = p_\theta(f^l(x))$. Similarly, for each mutated sample $m_j \in \mathcal{M}t$, we compute the predicted probability: $p(y_i \mid m_j) = p_\theta(f^l(m_j))$. We then measure the change in prediction confidence between the mutated sample and the seed input.

$$\Delta p_j = p(y_i \mid x) - p(y_i \mid m_j) \tag{3}$$

Mutated examples that cause larger reductions in prediction confidence are ranked higher (**Line 6**). By ranking based on Δp_j, DDPM filters out mutated examples that, despite showing substantial attention deviations, do not effectively reduce the model's confidence in the correct prediction.

4) Rank Fusion and Suspicious Example Selection: To combine the results from APM and DDPM, AdvSel employs Reciprocal Rank Fusion (RRF) [56]. RRF assigns each sample a score based on its ranks in both APM and DDPM, producing a combined ranking that reflects both the magnitude and direction of attention deviations:

$$\text{RRFscore}(x) = \sum_{t \in \{apm, ddpm\}} \frac{1}{\text{rank}_t(x) + p} \tag{4}$$

where p is a smoothing parameter to avoid division by zero.

In Algorithm 1, RRF computes the final ranking of mutations based on the fused scores from APM and DDPM (**Line 7**). The top-M samples are selected as suspicious mutations for querying $f^{(>l)}$ to obtain the prediction result of f (**Line 8**). If any selected sample results in a prediction different from that of the original example, it is identified as an adversarial example (**Lines 9-14**). Otherwise, the seed is updated to the mutated example that minimizes the victim model's prediction in the original label, and the process repeats (**Line 15**).

VI. AdvSel's Evaluation Design

To validate the effectiveness of AdvSel, we conduct experiments addressing the following RQs:

RQ4: Does AdvSel improve the cost-effectiveness of existing adversarial attack methods?

RQ5: Under resource-constrained attack scenarios, can AdvSel achieve better performance?

RQ6: How do individual components of AdvSel contribute to its overall effectiveness?

RQ7: How effective is the architectural design of DDPM compared with alternative variants?

RQ8: How does the choice of probing layer l influence the attack effectiveness of AdvSel?

A. Experimental Settings

1) Evaluation Datasets: To comprehensively evaluate our approach, we first assess its performance on the datasets listed in Section III-B3. To prevent overfitting, we generate adversarial examples using the test set of each dataset, distinct from the training data used in our empirical analysis. Additionally, to demonstrate AdvSel's generalizability, we conduct an evaluation on the Code Comment Inconsistency Detection (CCID) task. This task differs from prior ones in that it focuses on identifying semantic mismatches between code and its associated natural language comments. We select CCID to test whether AdvSel can generalize to tasks involving code–comment alignment and cross-modal understanding, which are becoming increasingly important in software engineering scenarios. We utilize the publicly available dataset provided in the replication package released by Xu et al. [57], [58], specifically selecting summary-type comments. This dataset consists of 10,498 examples, with 8,398 used for training and 2,100 for validation and testing. We attempt to reproduce the original results on these datasets, and the results indicate that our fine-tuned model achieves comparable performance.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

2) Victim Models and Baselines: To evaluate our method thoroughly, we first evaluate it on the models as listed in Section III-B1. Furthermore, GraphCodeBERT, UniXcoder, and PLBART are also evaluated as victim models to demonstrate ADVSEL's generalizability. Although these models may not reflect the most recent advancements in large-scale code LLMs, they remain standard baselines in the literature [12], [15], [59], [60], [61], and have been extensively studied in prior adversarial research. Their continued relevance enables direct and fair comparisons with existing black-box attack methods. Moreover, due to resource constraints and the high cost of evaluating large-scale models under adversarial settings, we prioritize models that are accessible and well-supported in the community. Importantly, the design of ADVSEL is model-agnostic and can be readily extended to newer and larger models. We leave the evaluation on larger-scale LLMs as promising future work. Since the focus of this work is on improving black-box attack methods, the baselines detailed in Section III-B2—ALERT, WIR-Random, and RNNS—are used directly. The selection of victim models aligns with prior studies [12], [15], [59], [60], [61] on adversarial attacks for code understanding tasks, ensuring comparability with existing research.

B. Implementations

We implement ADVSEL in Python, utilizing tree-sitter [50] for parsing. The parameter M for ADVSEL is set to $0.5 \times C$, where C represents the number of candidate identifiers in the original adversarial method. In our implementation, we set the layer $l = 0.5 \times L$ for constructing both the APM and DDPM, where L is the total number of layers in the victim model. For the majority of the victim models (CodeBERT, GraphCodeBERT, CodeT5, and UniXcoder), we designate the 6th layer as the probing layer ($l = 6$) consistent with their 12-layer depth. Similarly, for PLBART, we set $l = 3$ to accommodate its 6-layer encoder structure. This choice is based on empirical findings: all CLLMs exhibit significant differences in attention deviations between adversarial and failure examples at approximately half their network depth. While earlier layers could be selected in certain setups for efficiency, we consistently adopt the midpoint layer to maintain uniformity across all experiments. All experiments are conducted on a machine running Ubuntu 20.04.1 LTS, equipped with an Nvidia GeForce RTX 3090 GPU, an Intel XEON 6226R 2.90GHz CPU, and 32 GiB DDR4-3200 memory.

VII. ADVSEL'S EVALUATION RESULTS AND ANALYSIS

A. Answer to RQ4

1) Setup: we employ the following metrics to evaluate the attack's efficiency and effectiveness:

- **Effectiveness.** Following existing literature [12], [15], we evaluate the attack effectiveness using the Attack Success Rate (ASR). ASR represents the proportion of code snippet x in the test set $\mathcal{X}$ for which an attack method successfully generates an adversarial example x'. A higher ASR indicates superior effectiveness: $ASR = \frac{|x|f(x') \neq f(x), x \in \mathcal{X}|}{|\mathcal{X}|}$, where f represents the victim model.
- **Efficiency.** In line with prior studies [12], [15], we evaluate the efficiency using three two metrics: 1) Average Model Queries (AMQ): AMQ measures the average number of queries made to the victim model during adversarial example generation, directly reflecting the overall attack running time. 2) Average Running Time (ART): ART serves as a comprehensive metric for assessing the efficiency of the attack approach. For ADVSEL, the ART calculation includes not only the adversarial attack time but also the training time of the DDMP model.

2) Results: ADVSEL is designed to enhance the cost-effectiveness of existing adversarial attack methods by reducing computational overhead while maintaining attack effectiveness. While baseline methods provide a benchmark for ASR, our goal is for ADVSEL to achieve comparable ASR with significantly fewer computational resources. The evaluation results are summarized in Tables V and VI.

1) **Effectiveness:** As measured by ASR, ADVSEL delivers competitive performance, with only minor reductions compared to the baseline methods. On average, the absolute ASR difference is 0.70% for ALERT, 0.67% for WIR, and 0.62% for RNNS across all tasks and models. Notably, in some task-model configurations, ADVSEL even improves ASR. For example, VD&WIR&CodeT5 and CD&ALERT&UniXcoder show slight ASR gains when using ADVSEL, highlighting its ability to retain or even enhance attack performance in certain settings. This demonstrates that while ADVSEL focuses on efficiency gains, it can also retain the effectiveness of attacks in some scenarios, approaching the effectiveness of baselines.

2) **Efficiency:** ADVSEL exhibits substantial improvements. The AMQ metric shows a significant reduction across all models and tasks. For instance, in CodeT5, AMQ decreases by up to 49.09%, and similar trends are observed in GraphCodeBERT and CodeBERT, with reductions of up to 48.47% and 48.72%, respectively. Additionally, the ART results reinforce these efficiency improvements, with overall average performance increases ranging from 20.84% to 21.45% across the ALERT, RNNS, and WIR-Random methods. Specifically, ART, in VD under RNNS for CodeBERT, decreases by 60.97%, while in VD&ALERT&CodeT5, ART drops by up to 52.17%.

> **Summary**: ADVSEL enhances attack efficiency with minimal impact on effectiveness. On average, it reduces model queries by 34.98%–42.91% and runtime by 20.84%–21.45% across all settings, while maintaining similar ASR (only 0.62%–0.70% lower than the original methods). In certain cases (*e.g.,* CodeT5+WIR on VD), it even improves ASR. These results confirm that ADVSEL achieves cost-effective attacks without sacrificing adversarial performance.

B. Answer to RQ5

1) Setup: In resource-constrained conditions, ensuring the performance of the attack becomes a critical challenge. To evaluate ADVSEL under such settings, we simulate resource-constrained environments by limiting the number of queries

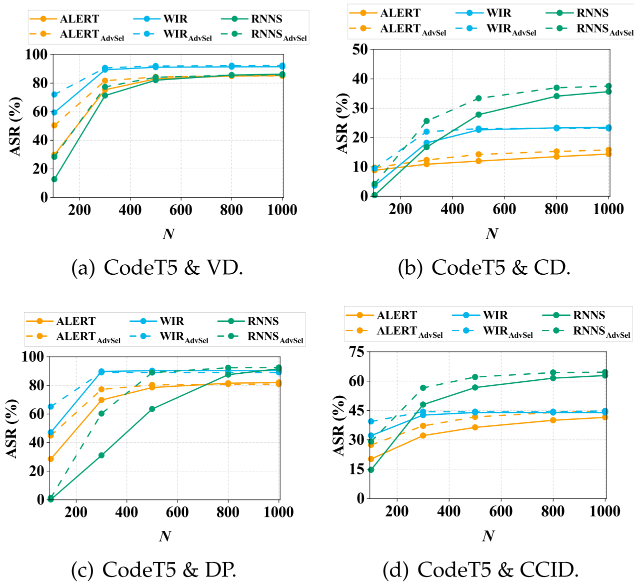
Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE V
THE ATTACK SUCCESS RATE (ASR (↑)) COMPARISON

Models	Tasks	ALERT (x)	+ ADVSEL (y)	Diff. ($y-x$)	WIR (x)	+ ADVSEL (y)	Diff. ($y-x$)	RNNS (x)	+ ADVSEL (y)	Diff. ($y-x$)
CodeBERT	**VD**	51.14	50.61	-0.53	56.88	56.35	-0.53	71.50	70.74	-0.76
	CD	22.81	22.99	0.18	33.14	32.70	-0.44	43.25	43.48	0.23
	DP	87.73	85.61	-2.12	89.98	88.15	-1.83	89.26	87.74	-1.52
	CCID	14.86	14.67	-0.19	22.29	22.67	0.38	36.57	37.33	0.76
GraphCodeBERT	**VD**	65.15	64.67	-0.48	62.54	62.60	0.06	72.14	71.01	-1.13
	CD	7.44	7.37	-0.07	23.47	23.13	-0.34	38.05	37.50	-0.55
	DP	95.56	94.79	-0.77	87.89	85.17	-2.72	94.32	92.70	-1.62
	CCID	41.37	40.26	-1.11	48.79	47.87	-0.93	60.48	60.48	0.00
CodeT5	**VD**	85.64	85.14	-0.50	91.49	92.36	0.87	86.99	85.88	-1.11
	CD	16.75	16.60	-0.15	25.08	24.79	-0.29	38.49	38.10	-0.39
	DP	82.35	80.85	-1.50	90.32	89.04	-1.28	93.35	92.50	-0.85
	CCID	47.24	46.48	-0.76	44.19	44.76	0.57	66.29	66.10	-0.19
PLBART	**VD**	76.60	75.69	-0.90	87.21	85.83	-1.39	90.89	90.65	-0.24
	CD	30.04	29.45	-0.59	35.78	35.34	-0.44	53.27	53.11	-0.15
	DP	88.31	86.43	-1.88	89.57	87.23	-2.34	93.75	91.68	-2.07
	CCID	71.32	70.14	-1.18	47.35	46.95	-0.39	74.07	73.87	-0.20
UniXcoder	**VD**	73.25	72.08	-1.17	74.77	73.95	-0.82	74.88	73.89	-0.99
	CD	8.84	9.59	0.75	28.61	28.17	-0.44	35.95	35.39	-0.57
	DP	61.88	60.81	-1.07	58.71	57.70	-1.02	33.34	31.82	-1.52
	CCID	33.46	33.46	0.00	58.06	58.06	0.00	70.86	71.24	0.38
Average	-	-	-	-0.70	-	-	-0.67	-	-	-0.62

made to the victim model. We evaluate ADVSEL against baseline methods by comparing their ASRs under these conditions. For each sample, the attack is allowed a maximum of N queries, where $N=\{100, 300, 500, 800, 1000\}$. If the attack fails after reaching the N queries, no further attempts are made on that sample, and the attack moves on to the next one. Due to space limitations, we report results using CodeT5 as the victim model, while the performance across other models is available in our replication package [26].

2) Results: Fig. 7 presents the ASR performance of different attack methods under varying query budgets. Different colors correspond to different attack methods, with solid lines indicating baseline methods and their dashed-line counterparts representing versions enhanced with ADVSEL. Across all tasks, increasing the query budget generally leads to higher ASR. However, under constrained query budgets, ADVSEL effectively enhances the ASR of black-box attack methods. This improvement is particularly pronounced in low-query scenarios, such as $N=100$ or $N=300$, where methods equipped with ADVSEL significantly outperform their baseline counterparts. When the query budget reaches $N=1000$, the ASR of most methods plateaus. ADVSEL offers significant benefits in query-limited settings but becomes less impactful as more queries are available, since attacks naturally achieve higher ASR with sufficient attempts. Despite the reduced ASR gains at higher query budgets, ADVSEL consistently improves attack efficiency. This efficiency advantage becomes more significant as the query budget grows, as shown in Table VI. These findings underscore the importance of query-efficient attack strategies, particularly in environments where computational resources are limited and excessive querying is impractical.



(a) CodeT5 & VD. (b) CodeT5 & CD. (c) CodeT5 & DP. (d) CodeT5 & CCID.

Fig. 7. ASR comparison within limited model queries.

Summary: ADVSEL maintains strong effectiveness under query constraints. For example, when limiting the number of model queries to 300, ADVSEL consistently outperforms baseline methods across all evaluated models and tasks. Specifically, it achieves an average attack success rate improvement ranging from 4.71% to 55.09% compared to the original attack methods. These results demonstrate that ADVSEL is particularly effective in resource-constrained adversarial settings.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE VI
THE EFFICIENCY COMPARISON (AMQ ↓ AND ART ↓)

Models	Tasks	Metrics	ALERT (x)	+ ADVSEL (y)	Diff. ($\frac{x-y}{x}$)	WIR (x)	+ ADVSEL (y)	Diff. ($\frac{x-y}{x}$)	RNNS (x)	+ ADVSEL (y)	Diff. ($\frac{x-y}{x}$)
CodeBERT	VD	AMQ	505.70	314.98	37.71%	164.28	110.96	32.46%	467.23	287.91	38.38%
		ART	0.2625	0.1321	49.67%	0.0693	0.0441	36.32%	0.2731	0.1066	60.97%
	CD	AMQ	1817.54	932.04	48.72%	221.56	128.40	42.05%	889.63	502.76	43.49%
		ART	0.9321	0.4744	49.10%	0.1038	0.0532	48.75%	0.5256	0.2334	55.59%
	DP	AMQ	294.63	201.02	31.77%	120.24	85.52	28.88%	444.45	281.38	36.69%
		ART	0.0437	0.0415	5.04%	0.0169	0.0159	5.92%	0.0672	0.0613	8.72%
	CCID	AMQ	889.76	464.80	47.76%	126.25	76.99	39.02%	573.83	324.42	43.46%
		ART	0.3836	0.3229	15.82%	0.0426	0.0382	10.24%	0.2072	0.1787	13.76%
GraphCodeBERT	VD	AMQ	588.55	329.97	43.94%	146.23	99.84	31.72%	474.52	292.08	38.45%
		ART	0.4342	0.3765	13.29%	0.0492	0.0476	3.25%	0.2560	0.2291	10.51%
	CD	AMQ	897.63	462.55	48.47%	235.83	136.12	42.28%	1088.59	594.02	45.43%
		ART	1.1828	1.1005	6.96%	0.2612	0.2220	15.01%	0.9676	0.7175	25.85%
	DP	AMQ	329.31	200.03	39.26%	131.58	91.93	30.13%	344.30	233.57	32.16%
		ART	0.0915	0.0748	18.25%	0.0327	0.0297	9.17%	0.0814	0.0793	2.58%
	CCID	AMQ	334.89	181.58	45.78%	113.64	70.70	37.79%	430.52	250.93	41.71%
		ART	0.3415	0.3342	2.13%	0.1697	0.1588	6.45%	0.4732	0.4533	4.20%
CodeT5	VD	AMQ	217.28	134.22	38.23%	99.28	75.37	24.09%	272.91	191.05	30.00%
		ART	0.1422	0.0680	52.17%	0.0598	0.0347	41.94%	0.1538	0.0921	40.11%
	CD	AMQ	2122.30	1080.54	49.09%	232.14	133.82	42.35%	774.88	442.80	42.86%
		ART	1.6826	1.0425	38.04%	0.2457	0.1275	48.10%	0.6658	0.4169	37.39%
	DP	AMQ	276.86	167.75	39.41%	123.52	86.43	30.03%	450.74	285.21	36.72%
		ART	0.0443	0.0426	3.84%	0.0200	0.0184	7.73%	0.0732	0.0677	-7.52%
	CCID	AMQ	686.69	373.58	45.60%	117.54	72.71	38.14%	374.73	225.30	39.88%
		ART	0.3004	0.2704	9.98%	0.0448	0.0406	9.43%	0.1536	0.1404	8.62%
PLBART	VD	AMQ	258.88	156.64	39.49%	115.59	85.43	26.09%	311.03	212.41	31.71%
		ART	0.0661	0.0400	31.99%	0.0296	0.0235	20.53%	0.0869	0.0629	27.65%
	CD	AMQ	1731.95	893.94	48.39%	215.14	126.72	41.10%	835.92	482.50	42.28%
		ART	0.7970	0.4332	45.65%	0.0943	0.0579	38.64%	0.4044	0.2401	40.62%
	DP	AMQ	257.03	159.47	37.95%	116.15	83.48	28.13%	351.94	235.47	33.09%
		ART	0.0198	0.0191	3.40%	0.0093	0.0089	4.08%	0.0289	0.0273	5.26%
	CCID	AMQ	457.49	263.73	42.35%	112.88	69.38	38.53%	284.46	177.64	37.55%
		ART	0.1772	0.1672	5.64%	0.0238	0.0223	6.50%	0.0605	0.0567	6.34%
UniXcoder	VD	AMQ	387.81	239.49	38.25%	138.77	97.63	29.64%	417.15	267.43	35.89%
		ART	0.1525	0.1000	34.33%	0.0429	0.0325	24.07%	0.1305	0.0999	23.47%
	CD	AMQ	2121.24	1084.17	48.89%	232.32	133.85	42.38%	1018.75	557.09	45.32%
		ART	0.6418	0.5266	17.94%	0.0661	0.0567	14.22%	0.3178	0.2622	17.50%
	DP	AMQ	656.31	394.85	39.84%	157.58	102.87	34.72%	927.70	522.27	43.70%
		ART	0.0958	0.0860	10.26%	0.0377	0.0201	46.73%	0.1446	0.1239	14.30%
	CCID	AMQ	582.00	307.07	47.24%	142.29	85.35	40.01%	584.06	358.88	38.55%
		ART	0.2622	0.2214	15.58%	0.0465	0.0373	19.78%	0.2095	0.1856	11.38%
Average		AMQ	-	-	42.91%	-	-	34.98%	-	-	38.87%
		ART	-	-	21.45%	-	-	20.84%	-	-	21.12%

C. Answer to RQ6

1) Setup: RQ3 investigates the contributions of each core component of ADVSEL, specifically the APM and the DDPM. We construct three variants of ADVSEL: 1) **w/o APM**: In this variant, instead of using attention deviation to rank mutations, we retain all other processes the same. 2) **w/o DDPM**: In this variant, the mutations are ranked based solely on the scores produced by the APM, omitting the directional evaluation of the deviations. 3) **ADVSEL$_{\text{random}}$**: This variant randomly selects M samples from the candidates for querying the CLLM, which provides a baseline for understanding the significance of the ranking strategies of ADVSEL. Due to page limitations, we adopt CodeT5 as the victim model since it shows the best performance in downstream tasks.

2) Results: The results in Table VII illustrate that both APM and DDPM significantly enhance ADVSEL's effectiveness. Removing APM generally reduces ASR, indicating its usefulness in ranking mutations based on attention deviation (*e.g.,* CodeT5&CD&WIR drops from 24.79% to 20.33%). The impact of DDPM removal is even more pronounced, with more frequent ASR declines compared to APM removal. This suggests that DDPM plays a crucial role in refining mutation ranking, substantially contributing to attack success rates across different tasks. Furthermore, when employing random

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE VII
ABLATION TEST FOR ADVSEL IN TERMS OF AVERAGE ASR

CLLM&Task	Attack	ADVSEL	w/o APM	w/o DDPM	ADVSEL$_{random}$
CodeT5&VD	ALERT	85.14	**85.33** (↑ 0.22%)	84.71 (↓ 0.51%)	82.06 (↓ 3.62%)
	RNNS	85.88	**86.50** (↑ 0.72%)	83.11 (↓ 3.23%)	84.53 (↓ 1.58%)
	WIR	**92.36**	90.75 (↓ 1.74%)	91.43 (↓ 1.00%)	85.33 (↓ 7.61%)
CodeT5&CD	ALERT	16.60	16.65 (↑ 0.31%)	**16.96** (↑ 2.19%)	16.18 (↓ 2.50%)
	RNNS	**38.10**	35.58 (↓ 6.60%)	38.04 (↓ 0.14%)	31.74 (↓ 16.68%)
	WIR	**24.79**	20.33 (↓ 17.99%)	23.83 (↓ 3.87%)	14.63 (↓ 41.00%)
CodeT5&DP	ALERT	**80.85**	79.05 (↓ 2.23%)	80.45 (↓ 0.49%)	77.73 (↓ 3.86%)
	RNNS	**92.50**	91.53 (↓ 1.05%)	92.21 (↓ 0.31%)	88.55 (↓ 4.27%)
	WIR	**89.04**	87.87 (↓ 1.32%)	88.71 (↓ 0.36%)	84.51 (↓ 5.08%)
CodeT5&CCID	ALERT	**46.48**	45.71 (↓ 1.64%)	45.52 (↓ 2.05%)	45.52 (↓ 2.05%)
	RNNS	**66.10**	**66.10** (↓ 0.00%)	64.76 (↓ 2.02%)	62.48 (↓ 5.48%)
	WIR	44.76	45.90 (↑ 2.55%)	44.00 (↓ 1.70%)	39.81 (↓ 11.06%)

mutation selection (ADVSEL$_{random}$), the performance degradation is especially severe. For instance, using random selection in the CodeT5&CD&WIR combination results in the ASR sharply declining from 24.79% to 14.63% (↓ 41.00%). A similar trend is observed in CodeT5&CCID&WIR, where ASR declines from 44.76% to 39.81% (↓ 11.06%). These findings emphasize the effectiveness of the proposed ranking strategies, as randomly selecting mutations significantly weakens attack performance.

Summary: Both APM and DDPM contribute to the overall effectiveness of ADVSEL. Ablation results show that removing either component degrades performance. For example, on the CodeT5&CD task with the WIR attack, removing APM results in a significant 17.99% drop in attack success rate (ASR), while removing DDPM leads to a 3.87% drop. These findings highlight the complementary roles of APM and DDPM: APM effectively filters out low-deviation mutations to reduce search space, while DDPM identifies directional shifts that drive misclassification. Together, they enable ADVSEL to achieve both high effectiveness and efficiency.

D. Answer to RQ7

1) Setup: To justify the architectural choice of DDPM, we evaluate a range of probe architectures that take the same FFN hidden representations from the victim model as input. The probe's task is to predict the classification label based solely on the hidden representation from the l-th layer, following a standard probing setup [25]. For each variant, we train the probe on FFN representations extracted from correctly predicted training samples and evaluate its accuracy on a validation set. A higher probe accuracy indicates a better ability to distinguish whether a mutation moves the model's internal representation toward or away from the correct label, which directly affects the quality of the deviation score Δp_j used for ranking mutations. We compare a diverse set of commonly used probe architectures, including logistic regression [62], support vector machines (SVMs) [63], convolutional neural networks (CNNs) [64], transformer-based probes [3], and the two-layer linear classifier adopted in DDPM. Due to space constraints, all experiments are conducted on CodeBERT using a vulnerability detection task.

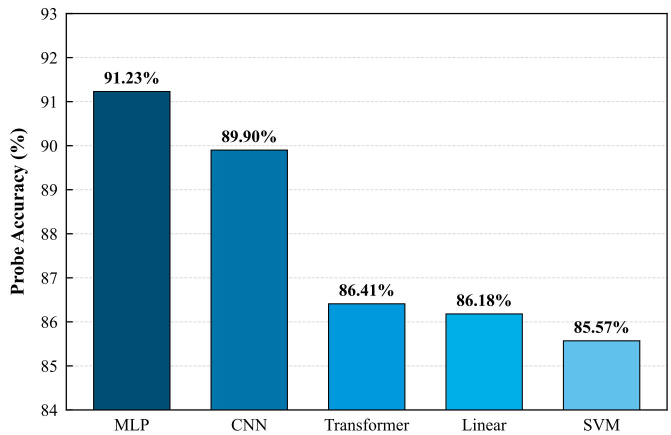


Fig. 8. Probe accuracy of different architectures on CodeBERT for the vulnerability detection task.

2) Results: As shown in Fig. 8, the MLP-based probe achieves the highest accuracy (91.23%) on the vulnerability detection task with CodeBERT, followed by the CNN-based probe (89.90%). More expressive architectures, such as transformer-based probes, do not provide additional benefits in this setting, while simpler models, including the linear classifier and SVM, exhibit lower accuracy. These results indicate that increasing architectural complexity does not necessarily translate into improved probe quality for this task. Instead, the two-layer linear design offers a favorable trade-off between predictive capability and simplicity, making it suitable for reliably estimating the deviation score Δp_j without introducing unnecessary overhead. This balance motivates our choice of a lightweight two-layer linear probe in DDPM.

Summary: The comparison shows that the two-layer linear probe is an appropriate choice for DDPM, offering better predictive reliability than the alternatives while keeping the model lightweight.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE VIII
IMPACT OF PROBING LAYER DEPTH ON ATTACK SUCCESS RATE (ASR) FOR VULNERABILITY DETECTION TASK. THE BEST RESULTS ARE HIGHLIGHTED IN BOLD

Layer Depth (l)	Attack Success Rate (ASR)		
	ALERT$_{+\text{AdvSel}}$	RNNS$_{+\text{AdvSel}}$	WIR$_{+\text{AdvSel}}$
Layer 3 (Shallow)	48.86%	69.87%	54.48%
Layer 6 (Middle)	**50.61%**	**70.74%**	56.35%
Layer 9 (Deep)	50.15%	70.10%	**56.47%**

E. Answer to RQ8

1) Setup: To assess whether selecting the middle layer ($l = 0.5 \times L$) is appropriate within our framework, we conduct an ablation study focusing on the vulnerability detection task using the CodeBERT model. Given that CodeBERT consists of $L = 12$ Transformer layers, we examine three representative probing depths: 1) a **shallow layer** (Layer 3, corresponding to $0.25 \times L$); 2) the **middle layer** (Layer 6, corresponding to $0.5 \times L$, our default setting); and 3) a **deep layer** (Layer 9, corresponding to $0.75 \times L$). For each configuration, we construct the corresponding APM and DDPM using the hidden states extracted from the specified layer, integrate them into ADVSEL, and execute the full adversarial attack pipeline on the same set of seed examples. Finally, we compare the Attack Success Rate (ASR) across these variants.

2) Results: The experimental results are summarized in Table VIII. We observe that the attack performance improves as the probing depth increases from the shallow layer (Layer 3) to the middle layer (Layer 6). However, as the depth further extends to the deep layer (Layer 9), the performance gain plateaus and exhibits diminishing returns. For instance, while Layer 9 achieves a marginally higher ASR on WIR (56.47%) compared to Layer 6 (56.35%), the improvement is negligible. Similarly, for ALERT and RNNS, the performance at Layer 9 is comparable to, or even slightly lower than, that at Layer 6. This trend underscores a critical trade-off between effectiveness and computational efficiency. Since our framework operates in a gray-box setting, obtaining attention features requires performing a forward pass through the model. Extracting features from deeper layers (*e.g.,* Layer 9) necessitates executing more Transformer blocks compared to the middle layer. Given that Layer 6 delivers robust attack performance nearly identical to that of deeper layers but requires significantly less computational overhead during feature extraction, we identify it as the most cost-effective configuration. Therefore, we select the middle layer as the default setting to maximize the speed of the attack pipeline without compromising success rates.

> **Summary**: The ablation results show that using the middle layer provides the reliable balance between attack success rate and query efficiency, supporting the use of $l = 0.5 \times L$ as the default probing depth in ADVSEL.

VIII. DISCUSSION

A. Time Complexity Analysis

Traditional methods evaluate all C candidate mutations through the full model, resulting in a time complexity of $O(C \times L \times T_L)$ per iteration, where L denotes the number of layers and T_L is the execution time per layer. In contrast, AdvSel fully evaluates only M candidates using the full model and processes the remaining $(C - M)$ candidates through only the first l layers ($l < L$). The total time complexity is thus reduced to:

$$O(M \times L \times T_L + (C - M) \times l \times T_L)$$

This significantly lowers the computational burden, especially when $M \ll C$ and $l \ll L$. Additionally, AdvSel introduces minimal overhead with DDPM, a lightweight two-layer linear classifier. Since its cost is negligible compared to transformer-based models, the overall complexity remains dominated by the term above. This efficiency gain becomes more pronounced with smaller values of M and l, as fewer candidates require full model evaluation and more are filtered using only partial forward passes.

B. Comparison With the White-Box Attack Method

We further compare ALERT augmented with AdvSel against CARROT [45], a representative white-box attack method that relies on gradient-guided search. CARROT is designed with full access to the victim model's internal parameters and gradients and is therefore expected to achieve strong attack performance. To ensure a fair comparison, we align our experimental setup with that used in CARROT's original study. Specifically, we conduct experiments on the GraphCodeBERT model using the defect prediction task, which serves as the primary evaluation setting in CARROT. All methods are evaluated on the same dataset. CARROT is executed using its official implementation [45], while ALERT and ALERT+ AdvSel are applied following the standard procedures described in this paper.

Table IX reports the comparison between ALERT augmented with AdvSel and the white-box attack CARROT on the defect prediction task using GraphCodeBERT. As expected, CARROT achieves a high attack success rate (ASR) of 95.59%, benefiting from full access to model parameters and gradients. Notably, ALERT attains a comparable ASR (95.56%), indicating that strong attack effectiveness can already be achieved in a black-box setting. When integrated with AdvSel, ALERT maintains a competitive ASR of 94.79%, with only a marginal decrease compared to both ALERT and CARROT. In contrast, ALERT+ AdvSel substantially improves attack efficiency. Specifically, its average model queries (AMQ) is reduced by 39.3% compared to ALERT (329.31 → 200.03), and is also significantly lower than that of CARROT. Moreover, ALERT+ AdvSel achieves the lowest average runtime (ART), even slightly outperforming the white-box baseline. These results demonstrate that, while CARROT benefits from white-box access, the combination of a black-box attack with AdvSel can achieve comparable attack effectiveness with substantially lower query cost and runtime.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE IX
PERFORMANCE COMPARISON WITH WHITE-BOX ATTACK (CARROT) ON DEFECT PREDICTION TASK USING GRAPHCODEBERT. THE BEST RESULTS ARE HIGHLIGHTED IN BOLD

Method	ASR (%)	AMQ	ART (s)
CARROT [45]	**95.59**	293.47	0.0757
ALERT	95.56	329.31	0.0915
ALERT + ADVSEL	94.79	**200.03**	**0.0748**

TABLE X
PRELIMINARY RESULTS OF INTEGRATING ADVSEL INTO THE WHITE-BOX ATTACK CARROT ON THE DEFECT PREDICTION TASK USING GRAPHCODEBERT

Metric	CARROT	CARROT + ADVSEL
AMQ	293.47	**172.89**
ART (s)	0.0757	**0.0677**
ASR (%)	**95.59**	94.77

AdvSel effectively bridges the efficiency gap between black-box and white-box attacks, offering a practical alternative in scenarios where gradient information is unavailable or access to the victim model is restricted.

C. Preliminary Exploration in the White-Box Setting

To explore whether the proxy-based filtering mechanism of AdvSel can extend its efficiency benefits to white-box settings, we conduct a preliminary experiment using CARROT. To ensure consistency with the comparative analysis in Section VIII-B, this experiment is strictly conducted on the defect prediction task using the GraphCodeBERT model. CARROT is a gradient-guided adversarial attack method that generates adversarial examples via iterative perturbation and optimization, relying on full access to model parameters. Methodologically, we integrate AdvSel into the CARROT pipeline as a candidate screening stage. Specifically, we apply AdvSel after CARROT produces a set of gradient-guided candidate mutations in each iteration. We then use the proxy signals provided by AdvSel to rank these candidates and only submit the top-ranked ones to the victim model for prediction. This design focuses the search on high-potential mutations, thereby accelerating the attack process while maintaining the guidance provided by white-box information.

Table X summarizes the preliminary results of integrating AdvSel into the white-box attack CARROT on the defect prediction task using GraphCodeBERT. Overall, the integration of AdvSel leads to a substantial improvement in attack efficiency, while only marginally affecting attack effectiveness. Specifically, CARROT+ AdvSel reduces the average number of model queries (AMQ) from 293.47 to 172.89, corresponding to a 41.1% reduction. This confirms that the proxy-based filtering mechanism of AdvSel effectively prunes low-potential mutation candidates before the gradient-based optimization step, thus avoiding a large number of unnecessary gradient evaluations. Consistently, the average runtime (ART) is also reduced from 0.0757s to 0.0677s, demonstrating that the reduced gradient workload translates into measurable runtime savings. In terms of attack effectiveness, CARROT+ AdvSel maintains a high attack success rate (ASR) of 94.77%, with only a minor decrease compared to the original CARROT (95.59%). Evidently, filtering a subset of mutation candidates does not substantially compromise the ability of the white-box attack to discover successful adversarial examples. In summary, these preliminary results highlight that the proxy-based filtering strategy of AdvSel is compatible with gradient-guided white-box attacks. Even when full model access is available, AdvSel can serve as an effective front-end to reduce computational overhead while largely preserving attack effectiveness.

D. Efficiency Analysis of Inference Mechanisms

To rigorously validate the computational efficiency of our proposed method and clarify the implementation details regarding the layer filtering mechanism, we conduct a comparative analysis of time cost under different inference strategies. A key concern in AdvSel is whether the computational savings are theoretically claimed or practically realized through efficient state management (*i.e.,* "freezing" and "resuming"). Therefore, we design an experiment to compare three distinct inference paradigms on the vulnerability detection task using CodeBERT, GraphCodeBERT, and CodeT5. Assuming a batch of mutation candidates is processed, the three strategies are defined as follows:

- **Full Inference (Original attack method):** All mutation candidates are fed into the model and undergo a complete forward pass from the input layer to the final output layer. No filtering is applied. This represents the standard black-box attack procedure.
- **Re-computation Strategy:** This strategy simulates a filtering mechanism without state caching. All candidates are first processed up to the probing layer to calculate attention deviation. After selecting the top-ranked candidates (top 50%), these selected samples are *re-input* into the model and processed from scratch to obtain the final prediction. This setting serves to evaluate the time cost if the "freezing" mechanism is not implemented.
- **Resume-based Strategy (ADVSEL):** This represents our actual implementation. All candidates are processed up to probing layer, where their intermediate hidden states are cached. After filtering, for the top-ranked candidates, we reload their cached states and *resume* the computation from the next layer to the final layer. This mechanism avoids redundant computations for the initial layers.

We record the average time required to process a single query sample under these three settings. Table XI reports a detailed comparison of inference time under three execution strategies: *Full Inference* (baseline), *Re-computation* (candidate filtering without state caching), and the proposed *Resume-based* strategy. The results highlight the critical role of intermediate

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

TABLE XI
COMPARISON OF AVERAGE INFERENCE TIME (SECONDS) UNDER DIFFERENT EXECUTION STRATEGIES. "FULL INFERENCE" IS THE BASELINE WITHOUT FILTERING; "RE-COMPUTATION" FILTERS BUT RESTARTS FROM SCRATCH (SIMULATING NO CACHING); "RESUME-BASED" IS OUR ADVSEL IMPLEMENTATION. **LOWER IS BETTER**

Model	**Full Inference** (Baseline)	**Re-computation** (w/o Cache)	**Resume-based** (**Ours**)
CodeBERT	0.2625	0.2994	**0.1321**
CodeT5	0.1422	0.0951	**0.0680**
GraphCodeBERT	0.4342	0.5501	**0.3765**

state reuse. For both CodeBERT and GraphCodeBERT, the *Re-computation* strategy incurs higher inference time than the baseline (*e.g.,* 0.2994s vs. 0.2625s for CodeBERT). This indicates that simply filtering candidates without caching intermediate representations is insufficient, as the additional overhead introduced by restarting forward computation outweighs the benefits of candidate reduction. In contrast, the *Resume-based* strategy adopted by AdvSel consistently achieves the lowest inference time across all evaluated models. Compared to full inference, it yields an approximately $2\times$ speedup on both CodeBERT and CodeT5, and remains clearly efficient on the GraphCodeBERT.

E. Threats to Validity

Internal Validity: A potential threat to internal validity lies in the possibility of bugs in our implementation. To address this, we conduct thorough code reviews and make our code and associated data publicly available, allowing for further verification by the research community. **External Validity:** The generalizability of our findings may be affected by the choice of tasks, datasets, and models. To mitigate this, we select widely recognized tasks and datasets, as used in prior state-of-the-art studies [12], [15], [60]. For victim models, we chose the most widely adopted CLLMs. In this work, we focus specifically on the more challenging task of code classification. As part of future work, we plan to extend our investigation to the domain of code generation.

IX. RELATED WORK

Adversarial example generation can be broadly categorized into black-box and white-box approaches [12], [15]. **Black-box Methods:** These approaches query model outputs to determine substitutions. ALERT [12] creates adversarial examples by replacing variable names using CLLM-generated candidates, while MHM [13] leverages the Metropolis-Hastings algorithm to sample identifier substitutions. STRATA [65] substitutes tokens based on token distributions, and Chen et al. [66] apply semantics-preserving transformations. CodeAttack [67] exploits strctural code features, and BeamAttack [15] perturbs source code using contextual information about identifiers. CODA [60] performs style transfers through structural transformations and identifier renaming based on training set references. **White-box Methods:** These methods rely on access to model gradients or architecture. CARROT [51] guides mutations using model gradients, while Henkel et al. [8] optimize attacks via gradient-based syntax tree transformations. Srikant et al. [68] apply optimized obfuscation, and DAMP [16] induces mispredictions by modifying inputs according to model gradients. **Comparison with ADVSEL**: Unlike abovementioned attack methods, which either operate solely in a black-box setting or require full gradient access in a white-box manner, AdvSel offers a gray-box approach. It leverages partial internal model information, specifically attention patterns, to improve attack efficiency. By using proxy models to assess attention deviations and their direction, AdvSel ranks and filters mutations, prioritizing those with higher adversarial potential, thereby reducing the need for excessive model queries.

X. CONCLUSION AND FUTURE WORK

We conduct an empirical study on the explainability analysis of adversarial attacks on CLLMs. Our findings show that adversarial examples induce significant model attention deviations, particularly in deeper model layers, compared to non-adversarial (failure) examples. We also observe that the direction of attention deviation plays a crucial role in the success of adversarial attacks. Based on these insights, we propose AdvSel, a framework that integrates with existing black-box attack methods to enhance efficiency without compromising effectiveness. Experimental results demonstrate that AdvSel maintains comparable attack success rates, with only a slight decrease of 0.62% to 0.70%, while significantly reducing model queries by 34.98% to 42.91%.

In future work, we plan to extend AdvSel to other large language models such as DeepSeek and StarCoder2, which feature more complex architectures and significantly larger parameter counts. This extension will help evaluate whether our proxy-based design remains effective when applied to models beyond the five CLLMs studied in this work. Moreover, while the current study focuses on code understanding tasks, such as vulnerability detection and code clone detection, a promising future direction is to investigate the applicability of AdvSel to generation-oriented tasks, including code summarization and automated code repair. Such tasks introduce different adversarial patterns, potentially requiring new forms of attention deviation analysis and proxy design.

ACKNOWLEDGEMENT

We appreciate the insightful insights provided by anonymous reviewers to improve the quality of the paper.

REFERENCES

[1] 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 *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2022, pp. 39–51.
[2] A. Fan et al., "Large language models for software engineering: Survey and open problems," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.: Future Softw. Eng. (ICSE-FoSE).* Piscataway, NJ, USA: IEEE Press, 2023, pp. 31–53.
[3] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing System*, 2017, vol. 30, pp. 6000–6010.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

[4] Z. Liu, Z. Tang, J. Zhang, X. Xia, and X. Yang, "Pre-training by predicting program dependencies for vulnerability analysis tasks," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

[5] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 2237–2248.

[6] W. Sun, M. Yan, Z. Liu, and D. Lo, "Robust test selection for deep neural networks," *IEEE Trans. Softw. Eng.*, vol. 49, no. 12, pp. 5250–5278, Dec. 2023.

[7] Z. Liu, Y. Feng, Y. Yin, J. Sun, Z. Chen, and B. Xu, "QATest: A uniform fuzzing framework for question answering systems," in *Proc. 37th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2022, pp. 1–12.

[8] J. Henkel, G. Ramakrishnan, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER).* Piscataway, NJ, USA: IEEE Press, 2022, pp. 526–537.

[9] Z. Li, G. Chen, C. Chen, Y. Zou, and S. Xu, "RoPGen: Towards robust code authorship attribution via automatic coding style transformation," in *Proc. 44th Int. Conf. Softw. Eng.,* 2022, pp. 1906–1918.

[10] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," in *Proc. 14th IEEE Conf. Softw. Test., Verification Validation (ICST).* Piscataway, NJ, USA: IEEE Press, 2021, pp. 36–46.

[11] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Inf. Softw. Technol.*, vol. 135, 2021, Art. no. 106552.

[12] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1482–1493.

[13] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, no. 01, pp. 1169–1176, 2020.

[14] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. Gall, "Adversarial robustness of deep code comment generation," *ACM Trans. Softw. Eng. Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–30, 2022.

[15] X. Du, M. Wen, Z. Wei, S. Wang, and H. Jin, "An extensive study on adversarial attack against pre-trained models of code," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 489–501.

[16] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[17] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014, *arXiv:1412.6572*.

[18] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *Proc. IEEE Symp. Security Privacy (SP).* Piscataway, NJ, USA: IEEE Press, 2016, pp. 582–597.

[19] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," 2017, *arXiv:1705.07204*.

[20] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," 2020, *arXiv:2002.08155*.

[21] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021, *arXiv:2109.00859*.

[22] Z. Wu, Y. Chen, B. Kao, and Q. Liu, "Perturbed masking: Parameter-free probing for analyzing and interpreting BERT," 2020, *arXiv:2004.14786*.

[23] E. Chiramal and K. S. B. Kai, "A grey-box text attack framework using explainable AI," 2025, *arXiv:2503.08226*.

[24] Y. Xu, X. Zhong, A. J. Yepes, and J. H. Lau, "Grey-box adversarial attack and defence for sentiment classification," 2021, *arXiv:2103.11576*.

[25] G. A., "Understanding intermediate layers using linear classifier probes," 2026, *arXiv:1610.01644*.

[26] Anonymous GitHub. 2024. Accessed: March 15, 2024. [Online]. Available: https://anonymous.4open.science/r/AdvSel-7231/

[27] B. Kou, S. Chen, Z. Wang, L. Ma, and T. Zhang, "Is model attention aligned with human attention? an empirical study on large language models for code generation," 2023, *arXiv:2306.01220*.

[28] S. Hooker, D. Erhan, P.-J. Kindermans, and B. Kim, "Evaluating feature importance estimates," 2018, *arXiv:1806.10758*.

[29] C. Molnar, *Interpretable Machine Learning: A Guide For Making Black Box Models Interpretable*, Morisville, NC, USA: Lulu, 2019.

[30] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, "What does BERT look at? An analysis of BERT's attention," 2019, *arXiv:1906.04341*.

[31] A. Galassi, M. Lippi, and P. Torroni, "Attention in natural language processing," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 10, pp. 4291–4308, Oct. 2021.

[32] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," 2016, *arXiv:1612.08220*.

[33] S. Vashishth, S. Upadhyay, G. S. Tomar, and M. Faruqui, "Attention interpretability across NLP tasks," 2019, *arXiv:1909.11218*.

[34] K. Zhang, G. Li, and Z. Jin, "What does transformer learn about source code?" 2022, *arXiv:2207.08466*.

[35] J. Zhang et al., "A black-box attack on code models via representation nearest neighbor search," 2023, *arXiv:2305.05896*.

[36] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proc. 19th Int. Conf. Mining Softw. Repositories,* 2022, pp. 608–620.

[37] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.,* 2022, pp. 519–531.

[38] J. Zhang, Z. Liu, X. Hu, X. Xia, and S. Li, "Vulnerability detection by learning from syntax-based execution paths of code," *IEEE Trans. Softw. Eng.*, vol. 49, no. 8, pp. 4196–4212, Aug. 2023.

[39] 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 *Advances in Neural Information Processing System*, vol. 32, 2019.

[40] FFmpeg, "FFmpeg," 2024. Accessed on October 5, 2024. [Online]. Available: https://www.ffmpeg.org/

[41] QEMU, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," 2024. Accessed on June 2, 2024. [Online]. Available: https://sites.google.com/view/devign

[42] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.

[43] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.* Piscataway, NJ, USA: IEEE Press, 2014, pp. 476–480.

[44] CodeChef, "CodeChef," 2024. Accessed on October 5, 2024. [Online]. Available: https://codechef.com/

[45] H. Zhang et al., "Towards robustness of deep program processing models—detection, estimation, and enhancement," *ACM Trans. Softw. Eng. Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–40, 2022.

[46] R. Sharma, F. Chen, F. Fard, and D. Lo, "An exploratory study on code attention in BERT," in *Proc. 30th IEEE/ACM Int. Conf. Program Comprehension*, 2022, pp. 437–448.

[47] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: visualising image classification models and saliency maps," 2013, *arXiv:1312.6034*.

[48] A. Shrikumar, P. Greenside, A. Shcherbina, and A. Kundaje, "Not just a black box: learning important features through propagating activation differences," 2016, *arXiv:1605.01713*.

[49] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *Proc. 34th Int. Conf. Mach. Learn.*, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 2017, pp. 3319–3328.

[50] Treesitter, "Treesitter," 2024. Accessed on October 5, 2024. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[51] B. Kim and F. Doshi-Velez, "Towards a rigorous science of interpretable machine learning," 2017, *arXiv:1702.08608*.

[52] F. Wilcoxon, *Individual Comparisons by Ranking Methods.* Berlin, Germany: Springer, 1992.

[53] M. Geva, R. Schuster, J. Berant, and O. Levy, "Transformer feed-forward layers are key-value memories," 2020, *arXiv:2012.14913*.

[54] D. Dai, L. Dong, Y. Hao, Z. Sui, B. Chang, and F. Wei, "Knowledge neurons in pretrained transformers," 2021, *arXiv:2104.08696*.

[55] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[56] G. V. Cormack, C. L. Clarke, and S. Buettcher, "Reciprocal rank fusion outperforms condorcet and individual rank learning methods," in *Proc. 32nd Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2009, pp. 758–759.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.

[57] Z. Xu et al., “Code comment inconsistency detection based on confidence learning,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 3, pp. 598–617, Mar. 2024.

[58] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, “Deep just-in-time inconsistency detection between comments and source code,” in *Proc. AAAI Conf. Artif. Intell.*, vol. 35, no. 1, pp. 427–435, 2021.

[59] Z. Li, C. Zhang, M. Pan, T. Zhang, and X. Li, “AACEGEN: Attention guided adversarial code example generation for deep code models,” in *Proc. 39th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2024, pp. 1245–1257.

[60] Z. Tian, J. Chen, and Z. Jin, “Code difference guided adversarial example generation for deep code models,” in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE).* Piscataway, NJ, USA: IEEE Press, 2023, pp. 850–862.

[61] S. Zhou, M. Huang, Y. Sun, and K. Li, “Evolutionary multi-objective optimization for contextual adversarial example generation,” *Proc. ACM Softw. Eng.*, 2024, vol. 1, no. FSE, pp. 2285–2308.

[62] C. M. Bishop and N. M. Nasrabadi, *Pattern Recognition and Machine Learning.* Berlin, Germany: Springer, 2006, vol. 4, no. 4.

[63] C. Cortes and V. Vapnik, “Support-vector networks,” *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.

[64] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 2002.

[65] J. M. Springer, B. M. Reinstadler, and U.-M. O’Reilly, “STRATA: Simple, gradient-free attacks for models of code,” 2020, *arXiv: 2009.13562*.

[66] P. Chen, Z. Li, Y. Wen, and L. Liu, “Generating adversarial source programs using important tokens-based structural transformations,” in *Proc. 26th Int. Conf. Eng. Complex Comput. Syst. (ICECCS).* Piscataway, NJ, USA: IEEE Press, 2022, pp. 173–182.

[67] A. Jha and C. K. Reddy, “CodeAttack: Code-based adversarial attacks for pre-trained programming language models,” in *Proc. AAAI Conf. Artif. Intell.*, vol. 37, no. 12, pp. 14892–14900, 2023.

[68] S. Srikant et al., “Generating adversarial computer programs using optimized obfuscations,” 2021, *arXiv:2103.11882*.

Authorized licensed use limited to: CHONGQING UNIVERSITY. Downloaded on June 08,2026 at 09:10:47 UTC from IEEE Xplore. Restrictions apply.