CARL: Unsupervised Code-Based Adversarial Attacks for Programming Language Models via Reinforcement Learning

KAICHUN YAO, Institute of Software, Chinese Academy of Sciences, China
HAO WANG, School of Computer Science, University of Chinese Academy of Sciences, China
CHUAN QIN, PBC School of Finance, Tsinghua University, China
HENGSHU ZHU, Computer Network Information Center, Chinese Academy of Sciences, China
YANJUN WU, Institute of Software, Chinese Academy of Sciences, China
LIBO ZHANG*, Institute of Software Chinese Academy of Sciences, China

Code based adversarial attacks play a crucial role in revealing vulnerabilities of software system. Recently, pre-trained programming language models (PLMs) have demonstrated remarkable success in various significant software engineering tasks, progressively transforming the paradigm of software development. Despite their impressive capabilities, these powerful models are vulnerable to adversarial attacks. Therefore, it is necessary to carefully investigate the robustness and vulnerabilities of the PLMs by means of adversarial attacks.

Adversarial attacks entail imperceptible input modifications that cause target models to make incorrect predictions. Existing approaches for attacking PLMs often employ either identifier renaming or the greedy algorithm, which may yield sub-optimal performance or lead to high inference times. In response to these limitations, we propose CARL, an unsupervised black-box attack model that leverages reinforcement learning to generate imperceptible adversarial examples. Specifically, CARL comprises a programming language encoder and a perturbation prediction layer. In order to achieve more effective and efficient attack, we cast the task as a sequence decision-making process, optimizing through policy gradient with a suite of reward functions. We conduct extensive experiments to validate the effectiveness of CARL on code summarization, code translation, and code refinement tasks, covering various programming languages and PLMs. The experimental results demonstrate that CARL surpasses state-of-the-art code attack models, achieving the highest attack success rate across multiple tasks and PLMs while maintaining high attack efficiency, imperceptibility, consistency, and fluency.

CCS Concepts: • **Security and privacy** → *Software security engineering*; • **Computing methodologies** → **Artificial intelligence**.

Additional Key Words and Phrases: Code Adversarial Attacks, Programming Language Models, Reinforcement Learning

1 INTRODUCTION

In recent years, there has been a remarkable advancement in the field of pre-trained language models for programming languages. These models have demonstrated exceptional capabilities in code representation learning

*Corresponding authors

Authors' addresses: Kaichun Yao, Institute of Software, Chinese Academy of Sciences, Beijing, China, yaokaichun@outlook.com; Hao Wang, School of Computer Science, University of Chinese Academy of Sciences, Beijing, China, wanghao184@mails.ucas.ac.cn; Chuan Qin, PBC School of Finance, Tsinghua University, Beijing, China, chuanqin0426@gmail.com; Hengshu Zhu, Computer Network Information Center, Chinese Academy of Sciences, Beijing, China, zhuhengshu@gmail.com; Yanjun Wu, Institute of Software, Chinese Academy of Sciences, Beijing, China, yanjun@iscas.ac.cn; Libo Zhang, Institute of Software Chinese Academy of Sciences, Beijing, China, libo@iscas.ac.cn.

ACM 1557-7392/2024/8-ARTx
https://doi.org/10.1145/3688839

and have demonstrated outstanding performance across a broad spectrum of downstream tasks. Notably, through extensive pre-training on expansive code corpora, a diverse range of programming language models (PLMs) such as CodeBERT [11], GraphCodeBERT [16], CodeT5 [49], and UniXcoder [15] have made significant contributions to automating various software engineering development tasks. These tasks encompass code understanding, including code search and defect detection, as well as code generation tasks such as code-to-code translation, code-to-code refinement, and code-to-natural language summarization. The advent of these PLMs has revolutionized the field, pushing the boundaries of what is achievable in software engineering automation.

However, the data-driven pre-training of the aforementioned models is derived from language models and treats code sequences as natural language. As a consequence, these models tend to place greater emphasis on expressive variable names and meaningful function names within the code, possibly neglecting the underlying functional structure of the code itself [20]. That also makes PLMs be vulnerable to adversarial attacks, only making subtle perturbations to model inputs will result in erroneous outputs from the target model. As shown in Figure 1, even a minor alteration in the input code, akin to a typo in natural language scenarios, can lead to code summarization models generating incorrect code summaries. Therefore, for the pre-trained PLMs, these attacks have the potential to uncover system vulnerabilities and assess the robustness of models, which holds great significance in the realm of software and system security. In our research, we aim to conduct a meticulous investigation into the robustness and vulnerabilities of pre-trained PLMs via adversarial attacks.

Following the work of CodeAttack [20], a successful code based adversarial attack should have three essential properties: minimal perturbations akin to spelling errors or synonym substitutions in natural language, code consistency maintaining the original input and coding style, and code fluency that preserves the user's understanding. However, previous adversarial attack models for natural language [23, 28] do not sufficiently meet these criteria. Recently, Zhou et al. [61] propose a simple identifier substitution approach to craft adversarial code snippet only for code comment generation task, and Jha et al. [20] propose a greedy-search based approach (i.e., CodeAttack) to generate adversarial code for pre-trained PLMs on multiple downstream tasks. However, the former approach exhibits limited effectiveness due to its simplistic attack method, while the latter approach encounters challenges with low efficiency at inference time and the potential to get stuck in local optima due to its reliance on greedy-search strategies. As shown in Figure 1, CodeAttack exhibits a notable drop in the performance of the target model, requiring up to 51 queries to achieve its objective. This indicates its low efficiency when it comes to attacking.

To address the above challenges, we propose, CARL, an effective and efficient unsupervised black-box attack model that utilizes reinforcement learning to generate imperceptible adversarial code samples and demonstrates the vulnerabilities of the state-of-the-art PLMs to code-specific adversarial attacks. Specifically, CARL consists of a programming language (PL) encoder and an adversarial perturbation prediction layer, which share same architecture and are initialized with a pre-trained masked CodeBERT [11]. Since we have no ground truth adversarial samples, we optimize CARL via policy gradient (PG) using a suite of reward functions, which are elaborately designed for meeting the requirement of attack effectiveness and attack quality. To validate the effectiveness of CARL, we conduct extensive experiments on multiple tasks (i.e., code summarization, code translation and code refinement) across different programming languages. Experimental results demonstrate the effectiveness and efficiency of CARL compared with state-of-the-art (SOTA) baseline.

The main benefits of CARL can be summarized as follows:

- Unsupervised: No labelled adversarial samples are required for model training.
- Fast inference: At test time, CARL only performs one-step perturbation prediction to generate an adversarial sample, which is much more faster than the SOTA baseline.
- Better performance: CARL outperforms CodeAttack with a large margin in terms of attack quality as well as attack effectiveness.

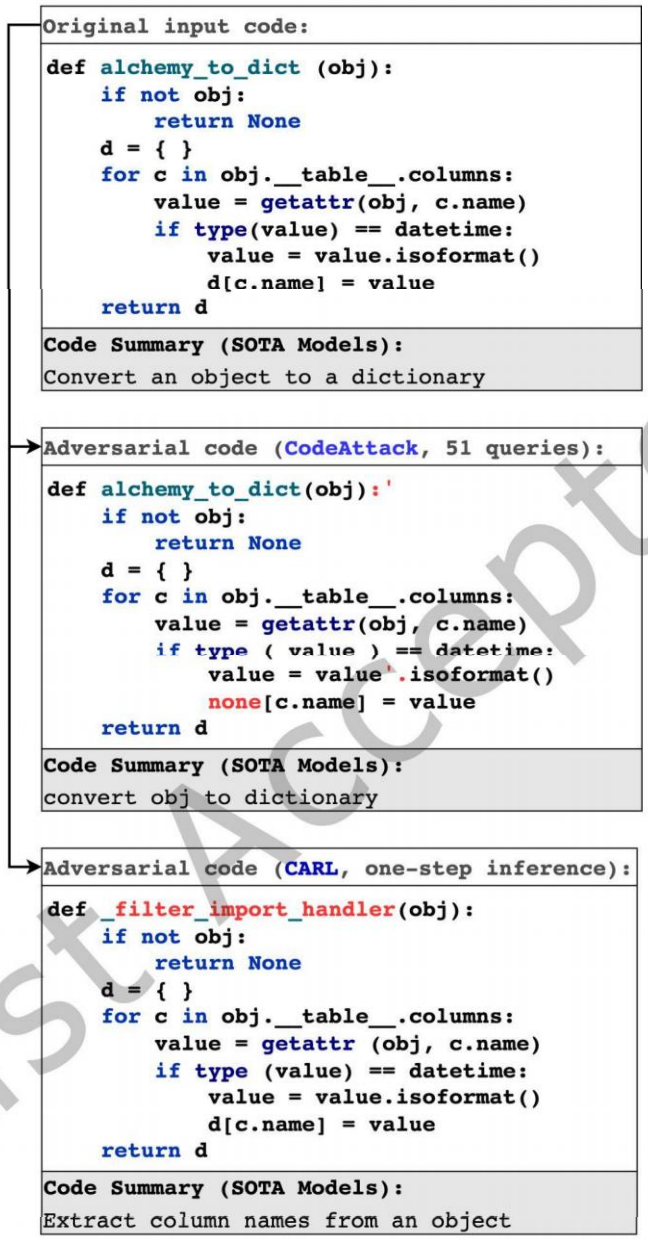

Fig. 1. The case examples for adversarial attack on Python code. The adversarial examples generated by CARL and CodeAttack. Small perturbations (red text) on the input python code will cause significant changes to the code summary obtained from the SOTA pre-trained PLMs.

2 RELATED WORK

In this section, we introduce the related work on pre-trained PLMs and adversarial attack.

2.1 Pre-trained PLMs

With the success of pretraining language models (LMs) like BERT [6] and GPT [39] in various applications [10, 21, 54], there has been a surge of research focused on LMs in the code domain in recent years. This has resulted in groundbreaking SOTA achievements across a wide range of code-related tasks. Code-based LMs, namely PLMs, are generally divided into three main architectures: encoder-only models, decoder-only models, and encoder-decoder models. Each of them is well-suited for specific tasks.

Encoder-only models excel in code understanding tasks like code retrieval. It usually adopts a bidirectional Transformer with self attention mechanism during pre-training, in which each token can attend to each other. For instance, CuBERT [25] leverages masked language modeling and next sentence prediction in BERT [6] as objectives to pre-train its parameters on a Python source code corpus, while CodeBERT [11] proposes a novel pre-training task called replace token detection to pre-train model on NL-PL pairs in six programming languages. Considering the inherent structure of code, GraphCodeBERT [16] use data flow in the pre-training stage to improve the capability of code representation, while SYN-COBERT [48] leverages abstract syntax trees (AST) edge prediction and contrastive learning as new objectives to augment representation learning. However, one limitation of encoder-only models is not suitable for generation tasks since they require an additional decoder for generation tasks. In these cases, the decoder must be initialized from scratch and cannot benefit from the pre-training for the encoder.

Regarding decoder-only models, as autoregressive language models continue to exhibit significant coding capabilities, researchers have recognized the benefits of continual pretraining on code data. Chen et al. [5] mark the advent of language models tailored for code with Codex, which are GPT-3 checkpoints pretrained on an additional 100 billion code tokens, representing one of the earliest multi-billion models designed specifically for code tasks. Further advancements include CodeLLaMA [43] training LLaMA 2 [46] on over 500 billion code tokens to develop Code LLaMA. Additionally, numerous specialized language models have emerged for code-related tasks. Models pretrained with causal language modeling (CLM), such as GPT-C [45], CodeGPT [32], CodeGen [35], PyCodeGPT [57], and CodeGeeX [60], have gained prominence in code processing. Some approaches successfully integrate denoising or multi-task pretraining with decoder architecture. Notable examples include Incoder [12], SantaCoder [2], and StarCoder [30], all trained with a fill-in-the-middle (FIM) objective. WizardCoder [33], employing the Evol-Instruct method, leverages Code Alpaca's 20K instruction-following code dataset to assemble a fine-tuning dataset of 78K instances with varying code instruction complexities. WizardCoder utilizes StarCoder [30] as the base model, fine-tuning it on the aforementioned instruction-based dataset. Decoder-only models excel in auto-regressive tasks, such as code completion and generation. However, the unidirectional framework proves sub-optimal for understanding tasks.

As for encoder-decoder models, they are suitable for handling both understanding and generation tasks. For instance, PLBART [1] adapts from the BART architecture [26] and utilizes the similar denoising objectives in BART to pre-train model on a combination of NL and PL corpora. Likewise, CodeT5 leverages the T5 architecture [40] and effectively incorporates essential token type information from identifiers, enabling multi-task learning for downstream tasks. Additionally, TreeBERT [22] follows the encoder-decoder transformer framework but takes advantage of the tree structural information by modeling AST paths. These encoder-decoder approaches hold great promise in effectively handling diverse programming language tasks.

Aiming to harness the benefits of the three aforementioned architectures, a unified pre-trained model named UniXcoder [15] is introduced for programming languages. UniXcoder adopts a UniLM-style design [7] and enables the support of diverse tasks by manipulating input attention masks. This approach amalgamates the

strengths of encoder-only, decoder-only, and encoder-decoder models, offering a versatile solution for various programming language tasks. In our study, we conduct a comprehensive investigation into the robustness and vulnerabilities of PLMs. To achieve this, we meticulously choose representative PLMs with diverse architectures, namely CodeBERT, GraphCodeBERT, CodeT5, and UniXcoder, as our target (victim) models to be subjected to adversarial attacks.

2.2 Adversarial Attacks

In recent years, deep neural networks (DNNs) have gained immense popularity in various artificial intelligence (AI) applications. Despite their success, studies have revealed that DNNs are susceptible to adversarial attacks. Adversarial examples are crafted with imperceptible perturbations, yet they have the capability to deceive DNNs and cause incorrect predictions. Due to the natural similarity between NLP and PL, this section focuses on the advancement of adversarial attacks in both domains.

2.2.1 Adversarial Attacks for NLP. Several existing methods employ different strategies to conduct adversarial attacks on NLP models. For instance, BERT based attack methods, i.e., BERT-Attack [29] and BAE [14], are proposed to discover and attack vulnerable words. TextFooler [24] and PWWS [41], on the other hand, leverage synonyms and part-of-speech (POS) tagging to replace crucial tokens. Deepwordbug [13] and TextBugger [27] adopt a character-based approach involving insertion, deletion, and replacement for attacks, while Hsieh et al. [18] and Yang et al. [51] implemented a greedy search and replacement strategy. Additionally, Alzantot et al. [3] employed a genetic algorithm, while Ebrahimi et al. [9] and Pruthi et al. [37] used model gradients to find substitutes for attacks.

2.2.2 Adversarial Attacks for PL. Compared to NL, PL have more structured nature. However, none of above methods have been specifically designed for programming languages. To investigate adversarial attacks on PL, researchers study adversarial attacks on specific software engineering tasks. For instance, Zhang et al. [59] employed Metropolis-Hastings sampling [34] to generate adversarial examples by renaming identifiers. To detect and measure the robustness of deep learning models on source code processing, Zhang et al. [58] proposed an optimization-based attack technique to generate valid adversarial code examples. To improve the natural semantic of examples, Yang et al. [52] generated the adversarial examples with a combination of heuristic method and genetic algorithm-based method. In addition, a variety of attack methods, such as gradient-based exploration [56] and metamorphic transformations [4, 17], are proposed to study the robustness of code based neural models. These models primarily focus on classification tasks like defect detection and clone detection. For generation tasks, Zhou et al. [61] proposed an identifier substitution approach to craft adversarial code snippets for code comment generation, and Jha et al. [20] proposed a code-specific greedy algorithm to generate adversarial code for pre-trained PLMs on generation tasks like code translation, code refinement, and code summarization. In our research, we also investigate the robustness of different pre-trained PLMs in generation tasks, aiming to overcome the shortcomings of greedy-search method and enhance the effectiveness of our study.

3 PRELIMINARIES

3.1 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning concerned with how agents learn to make sequential decisions in environments to achieve specific goals. In RL, an agent interacts with an environment by taking actions and receiving feedback in the form of rewards or penalties. The goal of the agent is to learn a policy, a mapping from states to actions, that maximizes cumulative rewards over time.

At the core of RL is the concept of the Markov Decision Process (MDP), which formalizes the interaction between an agent and its environment. An MDP consists of states, actions, transition probabilities, and rewards.

At each time step, the agent observes the current state of the environment, selects an action, and transitions to a new state according to the transition probabilities. The agent receives a reward based on the action taken and the resulting state transition. To learn an optimal policy, RL algorithms typically employ value functions or policy search methods. Value-based methods estimate the value of being in a particular state or taking a specific action, while policy search methods directly search for the best policy.

Reinforcement learning has been successfully applied to a wide range of problems, including game playing, robotics, decision making and planning [38, 53, 55]. In the adversarial attacks, the adversarial example generation can be regarded as a combinatorial problem. We aim at learning an optimal policy from huge action spaces with reinforcement learning to address this problem.

3.2 Adversarial Attacks

Adversarial attacks aim to find an adversarial perturbation δ for a given original input x, thereby generate an adversarial sample $x + \delta$ that can significantly undermines the performance of the model. Generally, adversarial attacks can be executed in two ways: white-box and black-box, depending on the attacker's level of knowledge about the model. In white-box attacks, the attackers possess complete access to the target model, including its architecture, parameters, training data, or loss functions. On the other hand, black-box attacks involve limited or no knowledge about the target model.

To thoroughly evaluate the vulnerabilities and robustness of existing pre-trained PLMs, we adopt a standard black-box access framework. In this scenario, the attacker lacks access to critical information about the target model. Instead, the attacker can solely interact with the pre-trained PLMs by submitting input sequences and observing the corresponding output results. Therefore, our study focus on a more practical assessment compared to a white-box scenario, where the attacker assumes unrestricted access to all the aforementioned components.

3.3 Problem Definition

Following the work of CodeAttack [20], code adversarial attacks can be formulated as follows: Given the input code $X \in \mathcal{X}$, the attacker identifies a perturbation δ to generate an adversarial sample X_{adv} that maximizes the degradation of the target model $F_t: \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are the input and output space, respectively. A valid adversarial attack satisfies the requirements for the input X:

$$X_{adv} = X + \delta \qquad s.t. \quad |\delta| \leq \theta \quad \& \quad \epsilon \leq \text{Sim}(X_{adv}, X) < 1, \tag{1}$$

where θ is the maximum allowed perturbation, $\text{Sim}(\cdot)$ denotes the similarity function and ϵ is the similarity threshold score.

According to the goal of the attacker, the generated adversarial sample X_{adv} and the original input X subject to the following conditions:

$$F_t(X_{adv}) \neq F_t(X), \tag{2}$$

$$Q(F_t(X)) - Q(F_t(X_{adv})) \geq \phi, \tag{3}$$

where $Q(\cdot)$ is the evaluation function measuring the quality of the output generated from the target model and ϕ denotes the quality drop. Finally, we formulate our final problem of generating adversarial samples as follows:

$$\Delta = argmax_{\delta}[Q(F_t(X)) - Q(F_t(X_{adv}))]. \tag{4}$$

The goal of code adversarial attacks is to discover minimal perturbations for X while adhering to constraints on the perturbation δ that maximize the disparity in quality $Q(\cdot)$ between the output sequences produced from the original input code X and the perturbed adversarial code X_{adv}.

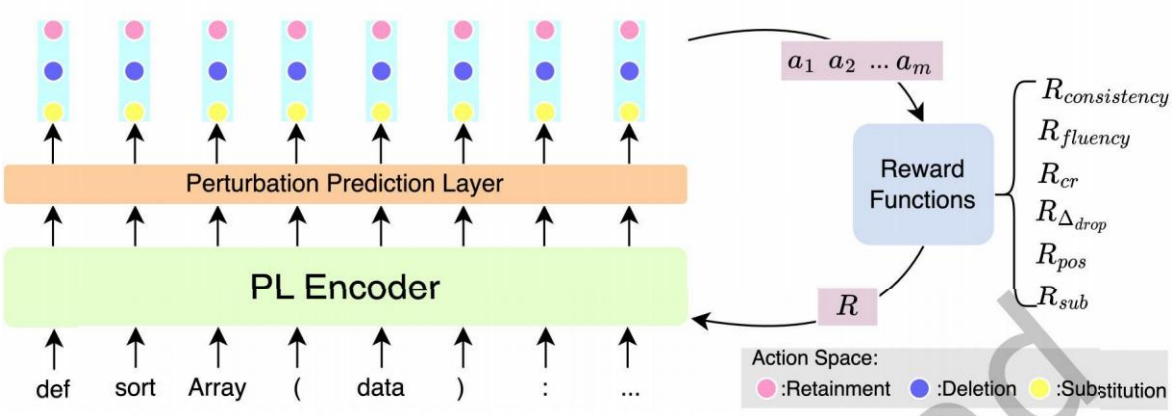

Fig. 2. The overview of our proposed CARL.

4 METHODOLOGY

In this section, we introduce the details of our CARL. Different from previous search based methods, CARL is an unsupervised code attack model with the parameter learnability, aiming at attacking the victim models across various programming languages and tasks. Figure 2 visualizes the design of CARL. The model architecture adopts a transformer encoder model with a linear classification layer for the perturbation prediction and we design a simple but effective reinforcement learning algorithm for the optimization of model parameters.

4.1 Model Architecture

4.1.1 PL Encoder. CARL accepts a sequence of code snippet as the input, and then predicts the perturbations to generate the output sequence. Recently, Transformer based pre-trained programming language models, such as CodeBERT [11], CodeT5 [49], and UniXcoder [15], show excellent performances on a variety of downstream tasks. To obtain the powerful ability of code understanding, we adopt a transformer encoder model to encode the code snippet, whose parameters are initialized with a pre-trained PLM, i.e., a masked CodeBERT model [11].

4.1.2 Perturbation Prediction. The perturbations reflect the changes within the input code sequence. Therefore, a key research problem is to concentrate on where to change and what to change in the original code. Current state-of-the-art method, i.e., CodeAttack [20], finds the most vulnerable tokens and then substitutes these vulnerable tokens to generate adversarial samples with a greedy search method. Different from CodeAttack, we consider the process of discovering and substituting the vulnerable tokens as a sequence decision-making problem, in which the perturbations on the code are regarded as action selections on the sequence. Specifically, we first use a RobertaTokenizer [31] tool to tokenize the input code X into a sequence of tokens $[x_1, x_2, ..., x_m]$. Then, we feed them to the PL encoder to obtain the corresponding semantic features $[h_1, h_2, ..., h_m]$. Finally, a linear classification layer followed by the PL encoder is used to predict the perturbation actions $A = [a_1, a_2, ..., a_m] \in \mathcal{A}$ based on each token in the input sequence.

$$[h_1, h_2, ..., h_m] = \text{PLE}([x_1, x_2, ..., x_m]), \tag{5}$$

$$P(a_i|X) = softmax(W_a h_i + b_a), \tag{6}$$

where PLE denotes the encoding process of PL encoder. W_a and b_a are the learnable parameters for the linear transformation.

In our scenario, the action space $\mathcal{A}$ encompasses three distinct types of actions: retainment, deletion, and substitution. Retainment signifies that the current token remains unchanged from the original token. Deletion implies the removal of the current token, while substitution entails replacing the current token with another token from the vocabulary V. In essence, the deletion and substitution actions can be likened to the typos that programmers might inadvertently make while writing code. It is worth noting that the PL encoder operates exclusively at the subtoken level, where each subtoken holds a distinct position. Our CARL have the ability to retain, delete, or substitute each subtoken, but insertion of new subtokens into the sequence is not permitted. Similar constraints are encountered in programming language model based methods such as CodeAttack [20] and ALERT [52]. These methods uses a masked PLM to identify vulnerable tokens and generate the top-k predictions for each masked token, creating an initial search space. The search method is then employed to iteratively substitute these vulnerable tokens with those from the search space. Throughout this process, no new tokens are inserted into new positions.

4.2 PG Optimization

Previous approaches regard adversarial example generation as a combinatorial problem. The greedy algorithm [20] [52] or genetic algorithm [52] are used to perform a search process. Different from these methods, we resort to reinforcement learning algorithm with learnable parameters to resolve this problem. Specifically, we train a policy π_θ with parameters θ to sample the action sequence A. Given an input X, the policy π_θ predicts an action probability distribution for each token index in X as follows:

$$\pi_\theta(A|X) = \prod_i P(a_i|X). \tag{7}$$

Actually, we do not have ground-truth action data to train the policy π_θ. Therefore, we train CARL using a policy gradient technique [44]. To simplify the learning setup, our policy π_θ is designed to perform a single action for a given input X and receives the corresponding reward immediately. In other words, we sample a sequence of actions $A = [a_1, a_2, ..., a_m]$ directly from π_θ each time and compute an immediate reward for the whole action sequence. It is different from representative sequential reinforcement learning scenarios that consider sequence sampling as multiple actions. Moreover, in our scenario, the policy can access multiple potential action sequences via sampling. Therefore, during training, we select the action sequence with the highest reward feedback by utilizing multiple samplings from the policy π_θ. Note that selecting the best action with the highest reward among several sampled actions enhances the training stability and expedites the convergence of CARL. Conversely, in typical sequential reinforcement learning scenarios, policies must execute a sequence of actions for a given input without immediate reward feedback for each action. Instead, a final delayed reward is received only after the last action is performed. This results in sparse reward signals for the actions, which can hinder training stability and convergence.

Finally, the training objective is to maximize the expected reward assigned to a predicted action sequence A for a given input X, which is computed using the reward function R:

$$\mathcal{J}(\theta) = E_{A\sim\pi_\theta(A|X)}[R(A)]. \tag{8}$$

Then, we use policy gradient theorem [44] with the REINFORCE algorithm [50] to compute the gradients, as follows:

$$\nabla_\theta \mathcal{J}(\theta) = E_{A_s\sim\pi_\theta(A_s|X)}[r_s \cdot \nabla_\theta log\pi_\theta(A_s|X)], \tag{9}$$

where $A_s \sim \pi(*|X)$ is a sample from the current policy at a given step, consisting of action sequence $A_s = (a_1^s, a_2^s, ..., a_m^s)$, and $r_s = R(X, A_s)$.

Following the common practice in using policy gradients, we subtract a baseline from the reward to reduce variance. In this case, we instantiate the baseline as $r_b = R(X, A_b)$, where A_b is the most likely action sequence according to the current policy. The gradient is then calculated as follows:

$$\nabla_\theta \mathcal{J}(\theta) = E_{A_s \sim \pi_\theta(A_s|X)}[(r_b - r_s) \cdot \nabla_\theta log \pi_\theta(A_s|X)]. \tag{10}$$

Thus, CARL is trained by minimizing the following loss function:

$$\mathcal{L}_\theta = E_{A_s \sim \pi_\theta(A_s|X)}[(r_b - r_s) log \pi_\theta(A_s|X)]. \tag{11}$$

By incorporating the baseline r_b, we can intuitively interpret that a sample A_s is encouraged if its reward (r_s) is higher than the current policy's prediction, resulting in a negative factor ($r_b - r_s$). Conversely, if the factor is positive, it indicates that the sample should be discouraged. The baseline serves as a reference point, guiding the model's learning process to improve the policy effectively.

4.3 Code-Specific Constraints

A flexible perturbations may hurt the syntactic correctness of the original code. Existing methods, like ACCENT [61] and CodeAttack [20], adopt heuristic rules to produce the restricted perturbations. Taking CodeAttack for instance, it maintains the code token class (i.e, identifiers, keywords, and arguments) alignment between original tokens and the potential substituted tokens. Different from these hard strategies, we leverage an extra PLMs, i.e. a new masked CodeBERT [11], as a benchmark, which forces the generated adversarial examples to maintain the code fluency and consistency. Specifically, given an original code sequence $X = [x_1, x_2, ..., x_m]$, CARL produces its corresponding adversarial code X_{adv} by sampling from the learned policy $\pi_\theta(*|X)$. In addition, the benchmark PLMs accepts the same input X and then outputs the initial policy distribution $\pi_{\theta_{init}}(*|X)$. Actually, the initial distribution $\pi_{\theta_{init}}$ are usually trained on a large-scale source code corpus that can maintain the code fluency and consistency to the maximum extent. Therefore, we use a penalty on KL divergence to make $\pi_\theta(*|X)$ approximate $\pi_{\theta_{init}}(*|X)$, which can be formulated as follows:

$$\mathcal{L}_{kld} = \mathrm{KL}(\pi_\theta(*|X), \pi_{\theta_{init}}(*|X)). \tag{12}$$

Finally, the loss function can be updated as:

$$\mathcal{L} = \mathcal{L}_\theta + \beta * \mathcal{L}_{kld}, \tag{13}$$

where β is a weight parameter, which is used to control the strength of the policy update diverging from the initial policy. The code-specific constraints leverage prior knowledge from pre-trained PLMs to guide perturbations, which can be regarded as the soft and flexible code-specific constraints. In our scenario, PLMs ensure that the generated code remains grammatically valid and preserves the semantics of the original code snippet, although occasional discrepancies may still occur. Therefore, we also introduce the concept of hard code-specific constraints as a variant of the soft code-specific constraints, implementing strict heuristic rules to limit perturbation. Specifically, we employed tree-sitter [1], a multi-language parser generator tool, to extract all variable names from syntactically valid code snippets. We mandate that perturbation predictions only apply to these variable names within the input code snippet to maintain operational semantics.

4.4 Reward Functions

We do not have direct access to ground-truth training data in our setup, so we design a suite of reward functions to achieve the goal of code adversarial attacks from two different aspects: *Attack Quality* and *Attack Effectiveness*. The former measures the quality of perturbations on the original input code and the later evaluates the impact of the the generated adversarial code on the victim models.

[1]https://tree-sitter.github.io/tree-sitter/

4.4.1 Attack Quality Rewards. As mentioned above, for the original input X, the perturbations δ must adhere to Equation 1. We design the reward functions from the following three aspects:

Code Consistency: It measures the consistency between the generated adversarial code X_{adv} and the original code X. The higher the value is, the more consistent the adversarial code is with the original source code. Formally, we use the CodeBLEU [42] metric to compute the reward score as follows:

$$R_{consistency} = \text{CodeBLEU}(X, X_{adv}). \tag{14}$$

Code Fluency: It measures the fluency of the adversarial codes. The higher the fluency score is, the more imperceptible changes the attacker makes. We use a masked programming language model (PLM) to estimate the fluency of an adversarial code. In particular, we calculate code fluency as the average logit of a token x^i_{adv} in the X_{adv} as follows:

$$R_{fluency} = \frac{1}{|X_{adv}|} \sum_{i=1} \text{PLM}(x^i_{adv}|X_{adv}). \tag{15}$$

Change Ratio: The goal of code adversarial attacks encourages producing the minimal perturbations, while no perturbations or too much perturbations will be punished. We cannot easily employ a hard constraint to control the number of changes on the original code. Instead, we impose a soft change ratio control using Gaussian reward functions. In particular, we use a reward function R_{cr} for the change ratio between the adversarial code and the original code. We set the distribution means of rewards μ as the desired values for change ratio, while the standard deviations are set as the mean times a factor η. In addition, we introduce a baseline reward b_s to ensure that R_{cr} becomes a punishment (i.e., a negative reward) when the change ratios diverge from the means too much.

$$R_{cr} = \mathcal{N}(\mu, (\eta \times \mu)^2) - b_s. \tag{16}$$

4.4.2 Attack Effectiveness Rewards. To make the adversarial perturbation dramatically degrade the target model's performance, we consider the reward functions as follows:

Performance Drop: It measures the performance drop of the victim model in the downstream task before and after the attack. We introduce the reward function for the performance drop as below:

$$R_{\Delta_{drop}} = \text{Q}(F(X), Y) - \text{Q}(F(X_{adv}), Y), \tag{17}$$

where $Q \in$ {CodeBLEU, BLEU} is the quality evaluation function and F is the pre-trained victim PLMs. CodeBLEU [42] measures the quality of the generated adversarial code X_{adv} for code translation and code refinement, and BLEU [36] measures the quality of the generated natural language summary when compared to the ground truth Y.

Attack Diversity: An empirical study shows that our CARL is inclined to produce the same or similar perturbations for different samples so as to achieve the maximal performance drop. Taking code summarization task for instance, the function names are easy to be attacked and the tokens in the function names are substituted into the other same tokens for different code samples. To prevent this phenomenon, we introduce a new reward function to measure the attack diversity in a batch of training samples. In other words, we encourage CARL learning a diverse attack policy that the tokens in different positions will be perturbed and substituted into the different tokens in each batch of samples. Specifically, we employ two hard constraints to control the attack diversity on the original code: position constraint function R_{pos} and substitution constraint function R_{sub}. Given a batch of training samples, we first simply divide each input sequence into n sections, uniformly. Then, we introduce a full zero vector $C \in \mathcal{R}^n$. At each training step, which section is changed by the action sampling from the policy, we plus 1 to the corresponding element. In addition, we also introduce a set K to record the substituted tokens. Finally, the perturbation diversity are formulated as follows:

$$R_{pos} = \begin{cases} 1 & \text{if sum}(C >= 1) > t_1, \\ -1 & \text{otherwise,} \end{cases} \tag{18}$$

$$R_{sub} = \begin{cases} 1 & \text{if len}(K) \ / \ \text{sum}(C) > t_2, \\ -1 & \text{otherwise,} \end{cases} \tag{19}$$

where len(·) is a function for computing the number of unique substituted tokens in K; sum(·) denotes the summation function for vector C; t_1 and t_2 are two threshold values for the position constraint and the substitution constraint, respectively.

The final reward is an average of the reward functions R_{pos}, R_{sub}, combined with $R_{consistency}$, $R_{fluency}$, R_{cr} and $R_{\Delta_{drop}}$:

$$R = \frac{1}{2}(R_{pos} + R_{sub}) + R_{consistency} + R_{fluency} + R_{cr} + R_{\Delta_{drop}}. \tag{20}$$

Finally, we provide a comprehensive outline of the training process of our CARL in Algorithm 1. For the inference of CARL, following completion of training, CARL begins by taking a source code X as input. Subsequently, CARL predicts token perturbations within X, thereby generating the adversarial example X_{adv}. This example can serve either to assess attack quality or to be input into victim models for evaluating attack effectiveness.

5 EXPERIMENT SETUP

5.1 Tasks and Datasets

Table 1. Statistics of datasets.

Task	Programming Languages	Dataset	
		Train	Test
Code Summarization	Python	14,000	15,000
	Java	5,000	11,000
	PHP	13,000	15,000
Code Translation	C#-Java	500	500
Code Refinement	Java	5,835	5,835

Algorithm 1 Training Process of CARL.

Input: The source code $X \in \mathcal{X}$, the training epoch K
Output: The policy parameters θ

```
Initialize the parameters of the PL encoder with a masked CodeBERT model;
for each i ∈ [0, K] do
    for each X ∈ 𝒳 do
        Calculate the token level semantic representations using Eq. (5);
        Perturbation prediction using Eq. (6);
        Sampling from the action sequence from the policy π_θ in Eq. (7);
        Calculate the reward using Eq. (20);
        Optimize the policy with Eq. (13);
    end for
end for
Obtain the learned policy paramters θ
```

We evaluate the performance of CARL on various PLMs based sequence-to-sequence tasks and programming languages. Specifically, we evaluate its performance in the following scenarios:

- Code Summarization: It aims at generating natural language summaries for provided code snippets. We select Python, Java, and PHP programming languages from the CodeSearchNet dataset. Since the train set has been used to train the victim models, we remove the train set in the CodeSearchNet dataset [19]. Thus, only the validation set and test set are used to train and test our CARL . Finally, we can obtain about 14,000/15,000, 5,000/11,000, 13,000/15,000 train/test samples for Python, Java and PHP, respectively.
- Code Translation: It entails translating one programming language to the other between C# and Java bidirectionally. The publicly available code translation dataset [2] comprises a total of 11,800 pairs of parallel functions, with 1,000 of them specifically designated for testing purposes. We divide the original testing dataset into train set and test set evenly since the train set has been used to train the victim models.
- Code refinement: It aims to automatically rectify bugs in Java functions. The publicly available code refinement dataset provides pairs of buggy Java functions as the source and their corresponding fixed functions as the target. For our experiments, we utilize a small dataset [47], which includes 46,680 training samples, 5,835 validation samples, and 5,835 test samples. Since the training samples have been used to trained the victim models, we only select validation samples and test samples are used our train set and test set, respectively.

The statistics of datasets are shown in Table 1.

5.2 Victim Models

For our experiments, we carefully select a set of representative methods as our victim models:

- CodeBERT [11]: It is a bimodal pre-trained model designed for both programming language (PL) and natural language (NL), utilizing a Transformer-based neural architecture. The model undergoes training through a hybrid objective function that integrates the replaced token detection pre-training task. This task involves detecting plausible alternatives sampled from generators. CodeBERT excels in learning general-purpose representations that effectively support various downstream NL-PL applications, including natural language code search and code documentation generation. Through the fine-tuning of model parameters, it exhibits versatility in handling code-code and code-NL tasks. This extends to tasks such as code summarization, code refinement, and code translation in our scenario.
- CodeT5 [49]: It stands as a unified pre-trained encoder-decoder transformer-based programming language (PL) model, strategically harnessing code semantics through an identifier-aware pretraining objective. The model adopts a cohesive framework that adeptly accommodates both code understanding and generation tasks, facilitating multi-task learning. In the fine-tuning process, we adhere to the parameters outlined in the CodeT5 paper [49], ensuring consistency across three generation tasks.
- UniXcoder [15]: It stands as a unified cross-modal pre-trained Programming Language Model (PLM), seamlessly blending the virtues of encoder, decoder, and encoder-decoder architecture through the manipulation of input attention masks. This innovative model employs mask attention matrices featuring prefix adapters, providing precise control over the model's behavior. By leveraging cross-modal content such as Abstract Syntax Trees (AST) and code comments, UniXcoder elevates code representation to a new level. The fine-tuning process involves applying consistent settings for hyper-parameters across three generation tasks, aligning with the methodology detailed in the UniXcoder paper [15].
- GraphCodeBERT [16]: It is graph based pre-trained PLM with encoder architecture, which strategically incorporates the inherent structure of the program, leveraging the data-flow representation. In our

[2]https://github.com/microsoft/CodeBERT/blob/master/GraphCodeBERT/translation/data.zip

scenario, we fine-tune the model across three downstream generation tasks by setting the maximal input length of GraphCodeBERT to 512 and following the same setting for other hyper-parameters in the GraphCodeBERT paper [16].

- CodeGeeX2 [60]: It is a open-source multilingual code generation model with 6B parameters, is built upon the ChatGLM2 [8] architecture trained on a more extensive dataset of code.

Furthermore, we report the detailed statistics on parameter size and architecture specifics for all victim models in Table 2. For the implementation specifics of victim models, for encoder-only architectures like CodeBERT and GraphCodeBERT aimed at sequence-to-sequence (Seq2seq) generation tasks, they are initialized from pre-training models and necessitate the incorporation of a new decoder. This decoder is trained from scratch using available data for various Seq2seq tasks to ensure effectiveness. In contrast, encoder-decoder models like CodeT5 and UniXcoder [3] can be directly trained on Seq2seq tasks after initialized from pre-trained models. As for decoder-only models, we fine-tune CodeGeeX2 for tasks such as code summarization, translation, and refinement using corresponding training data.

Table 2. The parameter sizes and architecture of the victim models.

Victim Models	Architecture	Params
CodeBERT	Encoder-only	120M
GraphCodeBERT	Encoder-only	120M
UniXcoder	Encoder-Decoder	125M
CodeT5	Encoder-Decoder	220M
CodeGeeX2	Decoder-only	6B

5.3 Baseline Models

We select CodeAttack [20], a most recent SOTA methods with well-designed greedy-search strategy, as our baseline. CodeAttack employs a pre-trained masked CodeBERT model [11] as its adversarial code generator. By harnessing the inherent structure of the code, it produces imperceptible yet impactful adversarial examples. We also employ another baseline, namely ALERT [52], which adopts a combination of the greedy search and the genetic algorithms. ALERT also utilizes a pre-trained model such as CodeBERT and GraphCodeBERT to generate natural adversarial examples.

Moreover, we also compare several variants of our CARL to examine the relative influences of different reward setups and code-specific constraints:

- CARL(w/ hard): A variant of CARL with hard code-specific constraints to constrain adversarial perturbations.
- CARL(w/o kld): A variant of CARL without using KL divergence in Equation 12 to constrain adversarial perturbations.
- CARL(w/o flu): A variant of CARL without using the code fluency reward in Equation 15.
- CARL(w/o cr): A variant of CARL without using the change ratio reward in Equation 16.
- CARL(w/o div): A variant of CARL without using the attack diversity reward in Equation 18 and Equation 19 .

5.4 Evaluation Metrics

Following the CodeAttack [20], we evaluate the adversarial codes generated from our CARL from two aspects, including *attack effectiveness* and *attack quality*, which are also mentioned in the reward function designs.

[3]Actually, UniXcoder can support encoder-only mode, decoder-only mode and encoder-decoder-mode by manipulating input attention masks.

Attack Effectiveness:

- Δ_{drop}: To assess the performance degradation across different downstream tasks, we select CodeBLEU [42] and BLEU [36] metrics (see Equation 17) in our experiments. CodeBLEU quantifies the quality of the generated code snippets in tasks such as code translation and code refinement, while BLEU measures the fidelity of the generated natural language code summaries by comparing them to the corresponding ground truth summaries.
- Success rate: The rate of successful attacks, as measured by Δ_{drop}, is computed to gauge the effectiveness of the adversarial attack. A higher value indicates a more *effective* adversarial attack.

Attack Quality:

- Queries: It reflects the efficiency of the attack model. Once one token has been perturbed in the original code, the attacker has to query the victim model and observe the performance drop. The efficiency of the attacker is determined by the average number of queries required per sample. Therefore, a lower average number of queries indicates a more efficient attacker.
- Perturbation: It is used to measure the attack's perceptibility by computing the average number of tokens altered to generate an adversarial code. A lower value indicates a more imperceptible attack.
- Consistency: The consistency of the adversarial code is evaluated using CodeBLEU(X, X_{adv}) (see Equation 14). A higher score indicates a greater level of consistency between the adversarial code and the original source code.
- Fluency: It is a soft metric to measure the syntactic correctness of the generated adversarial code, automatically (see Equation 15).

5.5 Implementation Details

CARL is implemented in PyTorch. We use the publicly available pre-trained masked CodeBERT model [11] to initialize the parameters of CARL. The action space $\mathcal{A}$ copies from the vocabulary of CodeBERT with two additional tokens for the deletion action and retainment action. It is noted that the substitution action signifies the token replacement in the vocabulary. In addition, the learning rate was set to 1e-5 during the CARL training period. The hyperparameter β is set to 0.5 for code summarization task and 0.6 for code translation and refinement tasks. For the reward for change ratio, we set the mean μ to 0.2, the factor η to 1.5, and the baseline reward b_s to 0.5. For computing the attack diversity reward, each input sequence is divided into 5 sections. The threshold value t_1 and t_2 are set to 2.0 and 0.5, respectively.

6 EXPERIMENT RESULTS AND ANALYSIS

In this section, we perform experiments to answer research questions related to the performance of adversarial attacks. We care about the balance between attack effectiveness and attack quality, the robustness of victim models, attack efficiency the transferability of CARLand the efficacy of reward functions, which are discussed by answering five research questions, respectively.

RQ1. What is the impact of CARL on the attack effectiveness and the attack quality across various downstream tasks and programming languages?

RQ2. What is the efficiency of the adversarial samples produced by CARL in comparison to the baseline model?

RQ3. How is the transferability of CARL among different victim models?

RQ4. What is the impact of different components on the performance of CARL?

RQ5. Is it possible to utilize adversarial examples as a defense means to fortify the robustness of the victim models?

Table 3. Experimental results on code summarization, code translation and code refinement tasks. The best result is in boldface.

Task	Victim model	Attack method	Attack Effectiveness				Attack Quality		
			Before	After	Δ_{drop}	Success%	#Queries	#Perturb	#CodeBLEU
Summarize (PHP)	CodeBERT	CodeAttack	30.24	18.3	11.94	70.16	638.18	**3.56**	74.5
		ALERT		20.23	10.01	66.83	462.43	4.22	75.12
		CARL		**8.79**	**21.45**	**86.3**	**1.0**	5.63	**76.0**
	CodeT5	CodeAttack	32.17	19.32	12.85	77.36	563.49	7.22	83.66
		ALERT		20.43	11.74	75.44	396.52	8.74	83.23
		CARL		**5.84**	**26.3**	**91.96**	**1.0**	**5.18**	**85.41**
	UniXcoder	CodeAttack	31.25	12.65	18.6	84.44	431.48	**4.9**	**80.45**
		ALERT		10.32	20.93	88.43	382.3	5.13	80.31
		CARL		**4.75**	**26.5**	**92.34**	**1.0**	5.53	79.93
Summarize (Python)	CodeBERT	CodeAttack	23.64	15.65	7.99	64.71	907.6	3.97	62.09
		ALERT		17.93	5.71	78.92	402.4	2.83	79.3
		CARL		**13.65**	**9.99**	**75.07**	**1.0**	**1.15**	**91.0**
	CodeT5	CodeAttack	22.97	13.69	9.28	78.06	387.6	6.44	85.53
		ALERT		12.78	10.19	79.54	269.7	7.43	84.82
		CARL		**10.11**	**12.86**	**80.54**	**1.0**	**5.64**	**86.8**
	UniXcoder	CodeAttack	25.24	13.01	12.23	82.77	371.2	4.77	79.3
		ALERT		14.32	10.92	81.73	294.6	5.32	77.4
		CARL		**10.25**	**14.99**	**84.36**	**1.0**	**2.69**	**83.54**
Summarize (Java)	CodeBERT	CodeAttack	24.27	15.96	8.31	67.35	903.17	3.88	62.72
		ALERT		14.98	9.29	64.3	429.2	2.94	68.92
		CARL		**12.25**	**12.02**	**78.21**	**1.0**	**1.29**	**90.0**
	CodeT5	CodeAttack	24.28	14.3	9.98	**78.9**	519.77	8.2	72.58
		ALERT		15.41	8.87	77.69	258.5	7.41	71.63
		CARL		**10.25**	**14.03**	78.33	**1.0**	**6.18**	**73.0**
	UniXcoder	CodeAttack	26.83	11.42	15.41	88.29	314.85	3.98	81.91
		ALERT		10.98	15.85	87.94	167.5	3.13	82.65
		CARL		**9.89**	**16.94**	**89.36**	**1.0**	**2.69**	**84.53**
Translate (C#-Java)	CodeBERT	CodeAttack	79.64	57.02	22.62	79.76	62.12	1.81	62.18
		ALERT		56.87	22.77	80.32	46.74	1.23	79.46
		CARL		**55.32**	**24.32**	**80.94**	**1.0**	**1.14**	**84.0**
	CodeT5	CodeAttack	87.69	58.73	28.96	79.26	70.61	2.72	59.85
		ALERT		62.84	24.85	76.25	28.96	2.05	56.39
		CARL		**53.58**	**34.11**	**89.25**	**1.0**	**2.11**	**63.02**
	GraphCodeBERT	CodeAttack	83.19	47.07	36.12	87.43	14.63	1.29	70.76
		ALERT		46.86	36.33	88.21	8.4	1.45	73.39
		CARL		**44.74**	**38.45**	**89.86**	**1.0**	**1.12**	**84.0**
Refinement (Java)	CodeBERT	CodeAttack	72.75	56.77	15.98	59.72	39.97	1.47	85.79
		ALERT		48.93	*23.82*	*76.3*	*18.64*	**1.24**	88.34
		CARL		**15.93**	**57.82**	**93.28**	**1.0**	1.84	**90.0**
	CodeT5	CodeAttack	71.43	58.82	12.61	47.05	63.59	1.89	89.15
		ALERT		61.82	9.61	15.89	35.67	1.98	87.53
		CARL		**55.04**	**16.39**	**69.66**	**1.0**	**1.86**	**90.1**
	GraphCodeBERT	CodeAttack	72.16	45.55	26.61	75.7	15.77	1.23	88.64
		ALERT		40.52	31.64	82.49	6.98	1.13	89.34
		CARL		**23.87**	**48.29**	**93.53**	**1.0**	**1.07**	**91.0**

RQ1. What is the impact of CARL on the attack effectiveness and the attack quality across various downstream tasks and programming languages?

To answer RQ1, we have conducted extensive experiments on three different tasks, i.e., code summarization, code translation and code refinement. Table 3 shows the main results of CARL . In our evaluation metrics, Δ_{drop} and *Success%* are used to assess the attack effectiveness, while *#Perturb*, *#CodeBLEU* and *fluency* are used to evaluate the attack quality. Next, we conduct a detailed analysis in terms of specific task and code fluency, respectively.

Task Specific Analysis: For code summarization task, we report the experimental results on three kinds of programming languages: PHP, Python and Java. For each programming language, three representative PLMs, i.e., CodeBERT [11], CodeT5 [49] and UniXCoder [15], are attacked. As shown in Table 3, we can observe that CARL achieves the best performance in terms of attack effectiveness as well as attack quality compared to the baseline. Specifically, for language PHP, CARL achieves an impressive attack success rate of 86.3%, 91.96% and 92.34% across three victim models. These results correspond to a remarkable improvement of 16.14%, 14.5% and 8.1% over CodeAttack, respectively. In addition, CARL gains the performance improvement in terms of Δ_{drop} by 9.51, 13.45 and 7.9 on three victim models compared to CodeAttack. Meanwhile, CARL achieves nearly similar perturbations and CodeBLEU score with CodeAttack. On the other two programming languages, CARL achieves the similar experimental results.

For code translation task, we report the experimental results on a C# to Java translation dataset. We also select three typical PLMs, i.e., CodeBERT [11], CodeT5 [49] and GraphCodeBERT [16], as the victim models. It can be observed that CARL obtains the better performance drop and higher success rate of attack on the three victims compared to CodeAttack. To be specific, in terms of Δ_{drop}, CARL achieves the performance improvement by 1.7, 5.15 and 2.33 for three victim models, compared with CodeAttack. Meanwhile, the success rate of CARL gains the improvement of 1.18%, 9.99% and 2.43% over CodeAttack. In terms of attack quality, CARL obtains 84.0, 63.02 and 84.0 CodeBLUE score and the number of perturbations only reaches 1.14, 2.11 and 1.12 on three victim models, respectively. It is also totally superior over CodeAttack.

For code refinement task, we report the experimental results on a Java to Java refinement dataset. We select three same victim models as the code translation task. It can be observed that CARL outperforms CodeAttack with a large margin in terms of attack effectiveness as well as attack quality.

For the baselines, both CodeAttack and ALERT leverage a masked pre-trained programming language model to generate potential candidate substitutes for vulnerable tokens within input code. As seen in Table 3, ALERT and CodeAttack demonstrate comparable performance, but ALERT performs slightly worse when the victim model is CodeT5. This discrepancy could be attributed to the fact that ALERT is limited to perturbing variables within the input code. In contrast, CodeT5 has been pre-trained on tasks like "Masked Identifier Prediction", where altering identifier names does not affect code semantics. This pre-training strategy helps the model defend against attacks involving variable name changes.

Overall, CARL achieves the best performance on attack effectiveness and attack quality compared to the baselines, in terms of different tasks and programming languages.

Attacks on Large PLMs. Compared to CodeBERT, UniXcoder, GraphCodeBERT and CodeT5, we also consider a large PLM with 6B parameters, namely CodeGeeX2. CodeGeeX2 is a multilingual code generation model with decoder-only architecture. Actually, it is not directly used for code summarization, code translation and code refinement tasks. we need to fine-tune CodeGeeX2 for tasks such as code summarization, translation, and refinement using corresponding training data. The experimental results are shown in Table 4. Compared to PLMs with smaller parameters, CodeGeeX2 is more challenging to attack, resulting in lower attack effectiveness and quality. This could be attributed to the fact that large PLMs possess a stronger semantic understanding of

programming languages, giving them the capability to correct perturbations based on context. This is analogous to minor typos in natural language sentences that do not hinder human comprehension.

Table 4. Attacks of large PLM on code summarization, code translation and code refinement tasks. The best result is in boldface.

Task	Victim model	Attack method	Attack Effectiveness				Attack Quality		
			Before	After	Δ_{drop}	Success%	#Queries	#Perturb	#CodeBLEU
Summarize (Python)	CodeGeeX2	CodeAttack	27.75	24.53	3.22	51.23	1202.3	6.22	51.38
		ALERT		23.04	4.71	53.21	523.44	5.53	52.68
		CARL		**22.83**	**4.92**	**55.73**	**1.0**	**4.98**	**54.11**
Translate (C#-Java)	CodeGeeX2	CodeAttack	90.45	76.57	13.88	53.51	110.98	2.76	68.47
		ALERT		64.93	11.64	51.79	43.27	2.79	67.33
		CARL		**75.2**	**15.25**	**55.61**	**1.0**	**1.86**	**72.34**
Refinement (Java)	CodeGeeX2	CodeAttack	80.22	67.47	12.75	48.27	65.32	1.89	79.33
		ALERT		68.4	11.82	46.79	33.1	1.76	80.84
		CARL		**61.59**	**18.63**	**52.77**	**1.0**	**1.64**	**85.01**

Human Evaluation: We invited 3 human annotators, familiar with the Python, Java and C# programming languages, to check the naturalness and grammatical correctness of the generated adversarial code. Specifically, we randomly selected 100 samples from the generated adversarial samples for code summarization (Python), code translation (C#), and code refinement (Java) tasks, respectively. Subsequently, three annotators were invited to assess the naturalness and grammar validity of these samples.

For the naturalness assessment, the annotators were asked to classify the codes as either original or adversarial. The experimental results are presented in Table 5. On average, 70%, 82%, and 89% of the codes generated by CodeAttack were classified as original; 79%, 89%, and 94% of the codes generated by ALERT were classified as original; while 76%, 85%, and 93% of the codes generated by CARL were classified as original. Regarding the assessment of grammatical correctness, three annotators were asked to check the grammatical correctness of adversarial codes. The experimental results shows that CodeAttack obtained 86%, 84%, and 88% correctness in the code summarization, code translation, and code refinement tasks, respectively. ALERT obtained 90%, 88%, and 91% correctness, while CARL achieved 88%, 85%, and 90% correctness.

Evidently, ALERT achieved the best results in terms of naturalness and grammatical correctness, primarily because ALERT employs stricter attack rules by only allowing variables in the code to be attacked or substituted. However, this also reduces the effectiveness of adversarial attacks. Our CARL achieved suboptimal results in two metrics but demonstrated better attack effectiveness compared to ALERT.

Code Fluency: Fluency reflects syntactic correctness and imperceptible changes on the adversarial codes generated by attack models. It also indicates a higher attack quality. We also utilize a pre-trained masked CodeBERT [11] to compute the code fluency metric between CARL and baselines. As shown in Figure 3, we observe that CARL obtains better fluency scores (0.89, 0.88, 0.88, 0.81, 0.77, 0.87 and 0.88) than CodeAttack (0.83, 0.76, 0.82, 0.79, 0.75, 0.85, 0.85) and ALERT(0.85, 0.77, 0.8, 0.8, 0.72, 0.84, 0.86) on multiple tasks. It signifies that the adversarial codes from CARL is more natural and fluent than CodeAttack and ALERT.

Table 5. Human evaluation on the naturalness and grammatical correctness of the adversarial codes. The best result is in boldface.

Task	Victim model	Attack method	Naturalness	Grammatical Correctness
Summarize (Python)	CodeBERT	CodeAttack	70%	86%
		ALERT	**79%**	**90%**
		CARL	76%	88%
Translate (C#-Java)	CodeBERT	CodeAttack	82%	84%
		ALERT	**89%**	**88%**
		CARL	85%	85%
Refinement (Java)	CodeBERT	CodeAttack	89%	88%
		ALERT	**94%**	**91%**
		CARL	93%	90%

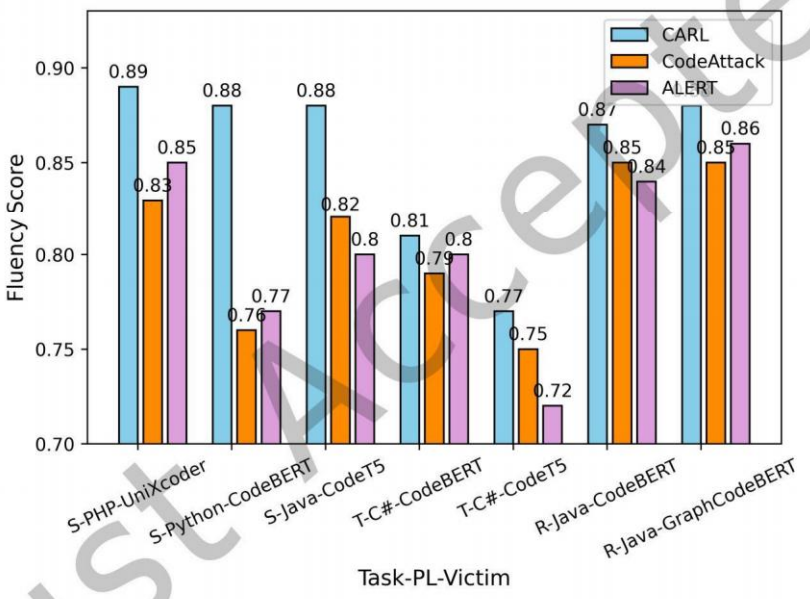

Fig. 3. Fluency score of CARL , CodeAttack and ALERT across different tasks, programming languages and victim models. "S", "T", "R" and "PL" are the abbreviation of code summarization, code translation, code refinement and programming language, respectively.

Answers to RQ1: Consistently, participants observe that the adversarial attacks generated by CARL outperform the current state-of-the-art baselines. This superior performance is evident in both attack effectiveness and attack quality, and it holds true across various downstream tasks and programming languages.

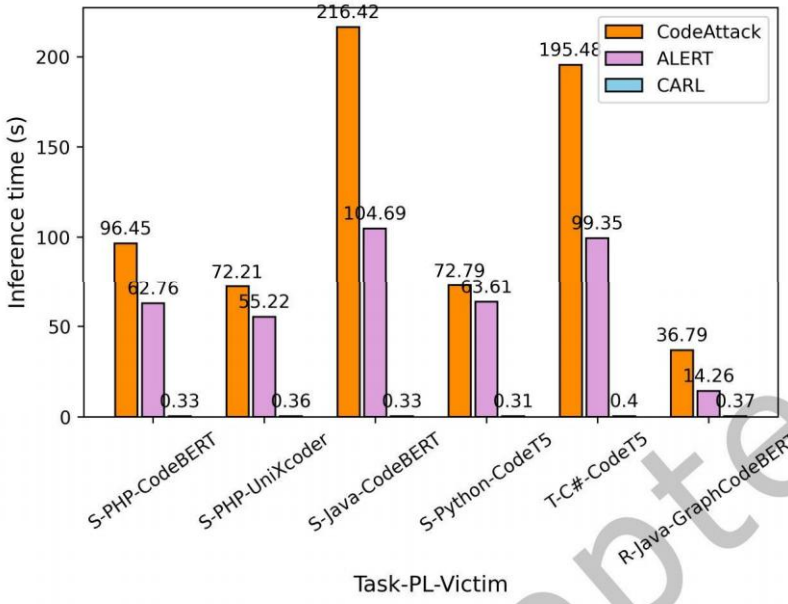

Fig. 4. The comparison of inference time between CARL , CodeAttack and ALERT across different tasks, programming languages and victim models.

RQ2. What is the efficiency of the adversarial samples produced by CARL in comparison to the baseline model?

To answer RQ2, we conduct the analysis of inference efficiency for the adversarial sample generation. Specifically, we randomly selected 1,000 samples from various tasks and reported the average inference time for each sample, as shown in Figure 4. We observe that CARL is significantly faster than CodeAttack and ALERT during inference. Notably, the metric *#Queries* is positively correlated with inference efficiency; more queries result in longer inference times. Our CARL requires far fewer queries than the other two baselines.

The primary reason for this efficiency is that baseline models must first identify vulnerable tokens in the original code and then iteratively substitute these tokens through a search method to generate adversarial code. In contrast, our CARL makes a one-step action decision to generate the adversarial code in a single pass, significantly enhancing inference efficiency.

Answers to RQ2: When we consider the metric #Queries and the average inference time for a single adversarial example, it is evident that CARL significantly outperforms CodeAttack and ALERT. In fact, it demonstrates a substantial improvement in inference efficiency, ranging from 100 to 600 times across various downstream tasks.

RQ3. How is the transferability of CARL among different victim models?

To answer RQ3, We conduct the analysis of model transferability. Generally, an adversarial example produced for a specific model is deemed transferable if it can effectively attack other victim models. To further investigate this research question, we first train CARL for attacking CodeBERT. Then, the trained CARL model is employed to

Table 6. Transferability study on code summarization, code translation and code refinement tasks. The term "transfer" signifies that CARL is trained using CodeBERT and then used to attack other victim models. On the other hand, its counterpart, referred to as "normal," signifies that CARL is trained and used to attack the same victim model. The best result is in boldface.

Task	Victim model	Attack Pattern	Attack Effectiveness				Attack Quality		
			Before	After	Δ_{drop}	Success%	#Queries	#Perturb	#CodeBLEU
Summarize (Java)	CodeT5	normal	24.28	**10.25**	**14.03**	**78.33**	1.0	**6.18**	**73.0**
		transfer		12.51	11.77	75.17		7.7	70.0
	UniXcoder	normal	26.83	**9.89**	**16.94**	**89.36**	1.0	**2.69**	**84.53**
		transfer		11.6	15.23	82.94		9.25	65.0
Summarize (Python)	CodeT5	normal	22.97	**10.11**	**12.86**	**80.54**	1.0	**5.64**	**86.8**
		transfer		14.69	8.28	69.75		6.79	72.0
	UniXcoder	normal	25.24	**10.25**	**14.99**	**84.36**	1.0	**2.69**	**83.54**
		transfer		13.33	10.92	78.06		7.86	69.0
Translate (C#-Java)	CodeT5	normal	87.69	53.58	34.11	89.25	1.0	**2.11**	**63.02**
		transfer		**42.54**	**45.15**	**92.83**		6.48	43.64
	GraphCodeBERT	normal	83.19	44.74	38.45	89.86	1.0	**1.12**	**84.0**
		transfer		**31.65**	**51.54**	**91.34**		5.05	57.30
Refinement (Java)	CodeT5	normal	71.43	55.04	16.39	69.66	1.0	**1.86**	**90.1**
		transfer		**33.76**	**37.67**	**88.63**		5.13	59.0
	GraphCodeBERT	normal	72.16	23.87	48.29	93.53	1.0	**1.07**	**91.0**
		transfer		**14.41**	**57.75**	**93.67**		1.85	87.3

perform attacks on CodeT5 and UniXcoder for the code summarization task, and on CodeT5 and GraphCodeBERT for the code translation and code refinement tasks. The experimental results are shown in Table 6.

From Table 6, it is evident that CARL achieves a decrease in performance for the code summarization task but exhibits performance improvement for the code translation and code refinement tasks in terms of attack effectiveness. In regards to attack quality, it demonstrates a decrease in both tasks. Despite a slight performance decrease, CARL still demonstrates exceptional transferability, highlighting its overall effectiveness in adversarial attacks. The primary reason for this may be that CARL learns an optimal policy for perturbation prediction, which provides strong generalization capabilities across different victim models.

Answers to RQ3: CARL exhibits strong transferability with small performance degradation, underscoring its remarkable capacity for generalization.

RQ4. What is the impact of different components on the performance of CARL?

To answer RQ4, we conduct the ablation study to evaluate the relative influences of different reward setups and code-specific constraints on performance of CARL. Specifically, we compare with several variants as we mentioned before on the code summarization, code translation and code translation tasks. The experimental results are shown in Table 7. From our observations, we note that CARL without KL divergence achieves the highest scores in terms of attack effectiveness but the lowest attack quality on both tasks. This suggests that using KL divergence as a code constraint can enhance attack quality but at the cost of attack effectiveness. Additionally, CARL with hard constraints improves attack quality due to stricter perturbation constraints, but this also negatively impacts attack effectiveness. Moreover, other variants either decrease attack effectiveness and improve attack quality, or vice versa. For example, CARL without flu and CARL without cr lack constraints on code fluency and change

Table 7. Ablation study on code summarization, code translation and code refinement tasks. The best result is in boldface.

Task	Victim model	Attack method	Attack Effectiveness				Attack Quality		
			Before	After	Δ_{drop}	Success%	#Queries	#Perturb	#CodeBLEU
Summarize (Java)	CodeBERT	CARL	24.27	12.25	12.02	78.21	1.0	1.29	90.0
		CARL w hard		11.04	13.23	76.72		1.13	91.23
		CARL w/o kld		**5.25**	**19.02**	**90.43**		3.1	70.32
		CARL w/o flu		9.65	14.62	81.36		1.78	84.5
		CARL w/o cr		10.33	13.94	80.38		1.65	88.34
		CARL w/o div		13.75	10.52	73.21		**1.01**	**92.37**
Translate (C#-Java)	CodeBERT	CARL	79.64	55.32	24.32	80.94	1.0	1.14	84.0
		CARL w hard		60.73	18.91	78.54		1.01	85.2
		CARL w/o kld		**50.21**	**29.43**	**86.32**		3.56	65.35
		CARL w/o flu		56.07	23.57	80.36		1.39	80.1
		CARL w/o cr		54.38	25.26	82.59		3.0	76.93
		CARL w/o div		61.49	18.15	76.77		**0.98**	**86.3**
Refinement (Java)	CodeBERT	CARL	72.75	15.93	57.82	93.28	1.0	1.84	90.0
		CARL w hard		17.84	54.91	89.33		1.66	91.02
		CARL w/o kld		**10.25**	**62.5**	**95.86**		4.63	71.33
		CARL w/o flu		16.87	55.88	92.05		2.32	86.3
		CARL w/o cr		15.29	57.36	92.73		2.87	82.36
		CARL w/o div		18.24	54.51	88.62		**1.56**	**91.24**

ratio, making the attack more flexible but reducing its quality. This analysis highlights the importance of carefully designed mechanisms or tricks to achieve an optimal trade-off between attack effectiveness and attack quality as an adversarial attack strategy.

It is worth noting that the hard code-specific constraints implement strict heuristic rules to limit perturbations, preventing parsing errors and other bugs. These rules, such as applying perturbation predictions only to variable names or substituting one operator with another, directly impact the quality of adversarial examples. However, these heuristic rules may not cover all cases.

To check the functions of these hard code-specific constraints, we conduct an experiment to report the error rate of adversarial examples using a parsing tool. Specifically, we randomly selected 200 examples from the testing dataset across the code summarization, code translation, and code refinement tasks. CARL then generated the corresponding adversarial codes, which were subsequently passed to different compilers for execution. The error rates, reported in Table 8, show that the error rate is controlled to below 7.5%. By analyzing the cases that result in errors, we can further adjust and optimize the heuristic rules.

Table 8. The error rate of adversarial code execution on code summarization, code translation, and code refinement tasks.

Task	Victim Models	Error Rate
Summarize (Python)	CodeBERT	7.5%
Translate (C#-Java)	CodeBERT	5.0%
Refinement (Java)	CodeBERT	6.0%

Answers to RQ4: Every component within CARL is crucial. The absence of any of these components will lead to a reduction in either attack quality or attack effectiveness.

Table 9. An analysis of robustness concerning victim models subjected to adversarial fine-tuning. "Before" refers to scenarios where victim models undergo no adversarial fine-tuning training, while "After" denotes the opposite condition.

Task	Victim model	Attack method	Success%		Λ_{drop}		#CodeBLEU	
			Before	After	Before	After	Before	After
Summarize (PHP)	CodeBERT-Adv	CodeAttack	70.16	54.3	11.94	6.32	74.5	61.3
		CARL	86.3	50.7	21.45	12.73	76.0	58.92
	CodeT5-Adv	CodeAttack	77.36	62.97	12.85	7.52	83.66	68.92
		CARL	91.96	70.32	26.3	17.76	85.41	61.69
	UniXcoder-Adv	CodeAttack	84.44	67.83	18.6	11.32	80.45	68.62
		CARL	92.34	70.45	26.5	14.43	79.93	61.84
Summarize (Python)	CodeBERT-Adv	CodeAttack	64.71	50.26	7.99	3.67	62.09	50.27
		CARL	75.07	51.64	9.99	4.21	91.0	72.4
	CodeT5-Adv	CodeAttack	78.06	61.84	9.28	4.36	85.53	64.87
		CARL	80.54	59.43	12.86	5.64	86.8	59.45
	UniXcoder-Adv	CodeAttack	82.77	58.93	12.23	6.8	79.3	68.35
		CARL	84.36	55.5	14.99	6.92	83.54	69.42
Summarize (Java)	CodeBERT-Adv	CodeAttack	67.35	53.54	8.31	4.32	62.72	51.32
		CARL	78.21	58.6	12.02	5.42	90.0	74.26
	CodeT5-Adv	CodeAttack	78.9	63.2	9.98	5.34	72.58	61.45
		CARL	78.33	59.2	14.03	6.87	73.0	58.28
	UniXcoder-Adv	CodeAttack	88.29	71.53	15.41	7.82	81.91	64.2
		CARL	89.36	65.35	16.94	6.39	84.53	57.82
Translate (C#-Java)	CodeBERT-Adv	CodeAttack	79.76	61.02	22.62	15.39	62.18	53.6
		CARL	80.94	57.42	24.32	12.64	84.0	66.24
	CodeT5-Adv	CodeAttack	79.26	59.47	28.96	20.45	59.85	48.32
		CARL	89.25	62.18	34.11	23.21	63.02	42.63
	GraphCodeBERT -Adv	CodeAttack	87.43	68.26	36.12	24.57	70.76	52.93
		CARL	89.86	64.35	38.45	21.4	84.0	58.28
Refinement (Java)	CodeBERT-Adv	CodeAttack	59.72	49.23	15.98	10.58	85.79	73.29
		CARL	93.28	72.93	57.82	42.72	90.0	71.38
	CodeT5-Adv	CodeAttack	47.05	38.91	12.61	6.38	89.15	76.34
		CARL	69.66	48.29	16.39	8.29	90.1	71.84
	GraphCodeBERT -Adv	CodeAttack	75.7	53.9	26.61	18.23	88.64	74.32
		CARL	93.53	72.48	48.29	28.61	91.0	72.28

RQ5. Is it possible to utilize adversarial examples as a *defense means to fortify the robustness* of the victim models?

In addressing this research question, we investigate the efficacy of adversarial fine-tuning as a defense mechanism against attacks. Our methodology involves utilizing CARL to generate adversarial examples for each victim model, tailored to their respective training sets. Samples where a victim model makes a significant erroneous prediction on an original input are excluded. For the remaining inputs in the training sets, we generate one adversarial example each. Specifically, if CARL successfully executes an attack, we select the adversarial example

by greedily sampling from the learned attack policy. In cases where our approach fails to launch an attack, we generate adversarial examples by sampling from the learned attack policies. We then choose the example that minimizes the victim model's confidence in the ground truth label. These generated adversarial examples are subsequently integrated into the original training set, forming the adversarial training set. The victim model is then fine-tuned using this augmented set.

Following adversarial fine-tuning, we derive four victim models: CodeBERT-Adv, CodeT5-Adv, UniXcoder-Adv, and GraphCodeBERT-Adv, subjecting them to evaluation across three tasks, i.e., code summarization, code translation, and code refinement. Table 9 presents a comparison of the new models' attack effectiveness and attack quality against previously generated adversarial examples. The reported metrics include the success rate (Success%) and the performance drop (Δ_{drop}) for attack effectiveness, and the consistency metric (#CodeBLEU) for attack quality.

From the observations in Table 9, it is evident that all the adversarially fine-tuned models exhibit significantly increased robustness. Simultaneously, when targeting these fine-tuned models, both CARL and CodeAttack experience a reduced success rate of attack, lower performance drop, and inferior attack quality. Notably, CARL performs inferiorly to CodeAttack across Success%, Δ_{drop}, and #CodeBLEU metrics. The potential reason is that the generated adversarial samples act as a form of data augmentation in traditional machine learning, increasing the diversity of observable samples and the number of vulnerable samples. Using these adversarial samples generated by CARL to augment the fine-tuning training set contributes to the enhanced robustness of victim models against CARL.

Answers to RQ5: The adversarial examples produced by CARL prove instrumental in enhancing the robustness of victim models. Fine-tuning victim models adversarially with examples generated by CARL has the potential to diminish the effectiveness and quality of attacks employed by adversarial methods.

7 DISCUSSION

In this section, we discuss the robustness of victim models when attacked, the parameter sensitivity in KL divergence based code-specific constraints and case study.

7.1 Model Robustness

The robustness of victim models can have a direct impact on the performance of attack model. Obviously, a more robust victim model is more difficult to be attacked by our CARL. We use the metrics, such as attack success rate, Δ_{drop} and queries, to reflect vulnerabilities of the victim models. As shown in Table 3, we can conclude that UniXCoder may be the most vulnerable model in terms of code summarization task due to its larger performance drop, higher attack success rate and less queries on the three programming language than other two victim models. On the contrary, CodeBERT is more robuster than other two victims. On the code translation task and code refinement task, GraphCodeBERT may be the most vulnerable, while CodeBERT is still more robust. CodeBERT's pre-training on tasks like "Masked Identifier Modeling" and "Replaced Token Detection" enables it to withstand attacks involving changes to identifier names, as such alterations have minimal impact on code semantics. On the other hand, GraphCodeBERT relies on data flow graphs in its pre-training, which involves predicting relationships between identifiers. Since CARL modifies identifiers and disrupts these relationships, it proves exceptionally effective against GraphCodeBERT. Consequently, GraphCodeBERT experiences a more pronounced decrease compared to other models when subjected to CARL attacks.

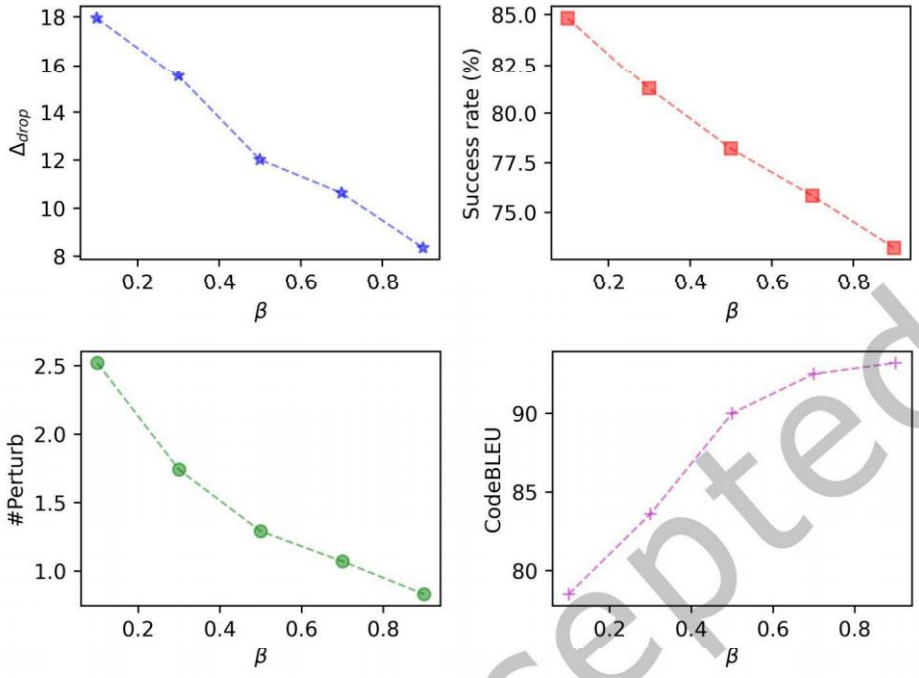

Fig. 5. The analysis of parameter sensitivity for varying β (PL: Java, Victim: CodeBERT, Task: Summarization).

7.2 Parameter Sensitivity

We conduct a sensitivity analysis of the weight parameter β in the KL divergence to validate the importance of code-specific constraint, which plays a critical role in both attack effectiveness and attack quality. As depicted in Figure 5, we observe that smaller values of β lead to higher Δ_{drop}, success rates, and perturbations, but lower CodeBLEU. This indicates that smaller β values improve attack effectiveness while reducing attack quality. Conversely, larger β values result in lower Δ_{drop}, success rates, and perturbations, but higher CodeBLEU. Hence, finding an appropriate β allows us to strike a balance between attack effectiveness and attack quality.

7.3 Training Cost

CARL is trained using feedback signals from the victim models, which make the training cost closely tied to both the parameter quantity of the victim models and the size of the training dataset. In Table 10, we provide the time of training convergence for CARL on various victim models across their respective tasks, where CARL is trained on a server with 4 NVIDIA A100 GPUs with 40G memory. It is evident that the training cost rises with an increase in the model's parameters. This is because larger victim models require more time to predict outputs based on the same input compared to smaller victim models. Note that although CodeBERT and GraphCodeBERT have the same size of parameters, GraphCodeBERT costs more time to handle the data flow graph of the input code compared to CodeBERT.

Table 10. The training time for CARL on different victim models.

Task	Training Data Size	Victims	Params	Training Time(h)
Code Summarization (PHP)	13,000	CodeBERT	120M	4.2
		UniXcoder	125M	4.3
		CodeT5	220M	5.1
		CodeGeeX2	6B	46.2
Code Summarization (Python)	14,000	CodeBERT	120M	4.8
		UniXcoder	125M	5.0
		CodeT5	220M	5.5
		CodeGeeX2	6B	48.6
Code Summarization (Java)	5,000	CodeBERT	120M	2.1
		UniXcoder	125M	2.2
		CodeT5	220M	2.8
		CodeGeeX2	6B	13.5
Code Translation (C#-Java)	500	CodeBERT	120M	0.5
		GraphCodeBERT	120M	0.6
		CodeT5	220M	0.65
		CodeGeeX2	6B	4.3
Code Refinement (Java)	5,835	CodeBERT	120M	2.3
		GraphCodeBERT	120M	2.5
		CodeT5	220M	3.1
		CodeGeeX2	6B	14.2

7.4 Case Study

To further illustrate the effectiveness of CARL, as shown in Figure 6, we present three adversarial code examples generated by the baseline CodeAttack and CARL. Specifically, the first example shown in the first row is for the code summarization task on the Python language, the second example in the second row is for the code translation task on the C# language and the third example in the third row is for the code refinement tasks on the Java language. For the first example in the Figure 6, we observe that adversarial examples generated by CARL exhibit greater naturalness and imperceptibility compared to those produced by CodeAttack. This superiority stems from CARL's learning mechanism, which aims to maximize reward scores concerning both code fluency and consistency between the original and adversarial code. Additionally, we leverage a pre-trained programming language model to impose constraints on perturbations, referred to as code-specific constraints. Moreover, our observations indicate that CARL tends to perturb function and variable names as part of its attack strategy. Unlike some existing methods such as ALERT, which restrict perturbations solely to identifiers within the original code, CARL adopts a more flexible approach while still achieving similar objectives through policy learning. However, this flexibility may lead to the generation of adversarial samples with invalid syntax due to the absence of strict heuristic rules like those governing restricted perturbations on identifiers.

8 THREATS TO VALIDITY

Internal validity: The results obtained in our experiment can vary depending on different hyperparameter settings, such as input length, the number of training epochs, the number of action sequence samples, and so on. To address these concerns, we standardize the input length for CodeBERT, GraphCodeBERT, CodeT5, and UniXcoder at 256 tokens, ensuring consistency in the number of tokens processed for the same code snippet. Regarding hyperparameters like β, μ, the factor η, and the learning rate, we employed a random search approach to identify suitable values. Different hyperparameters can significantly affect the model's performance. For

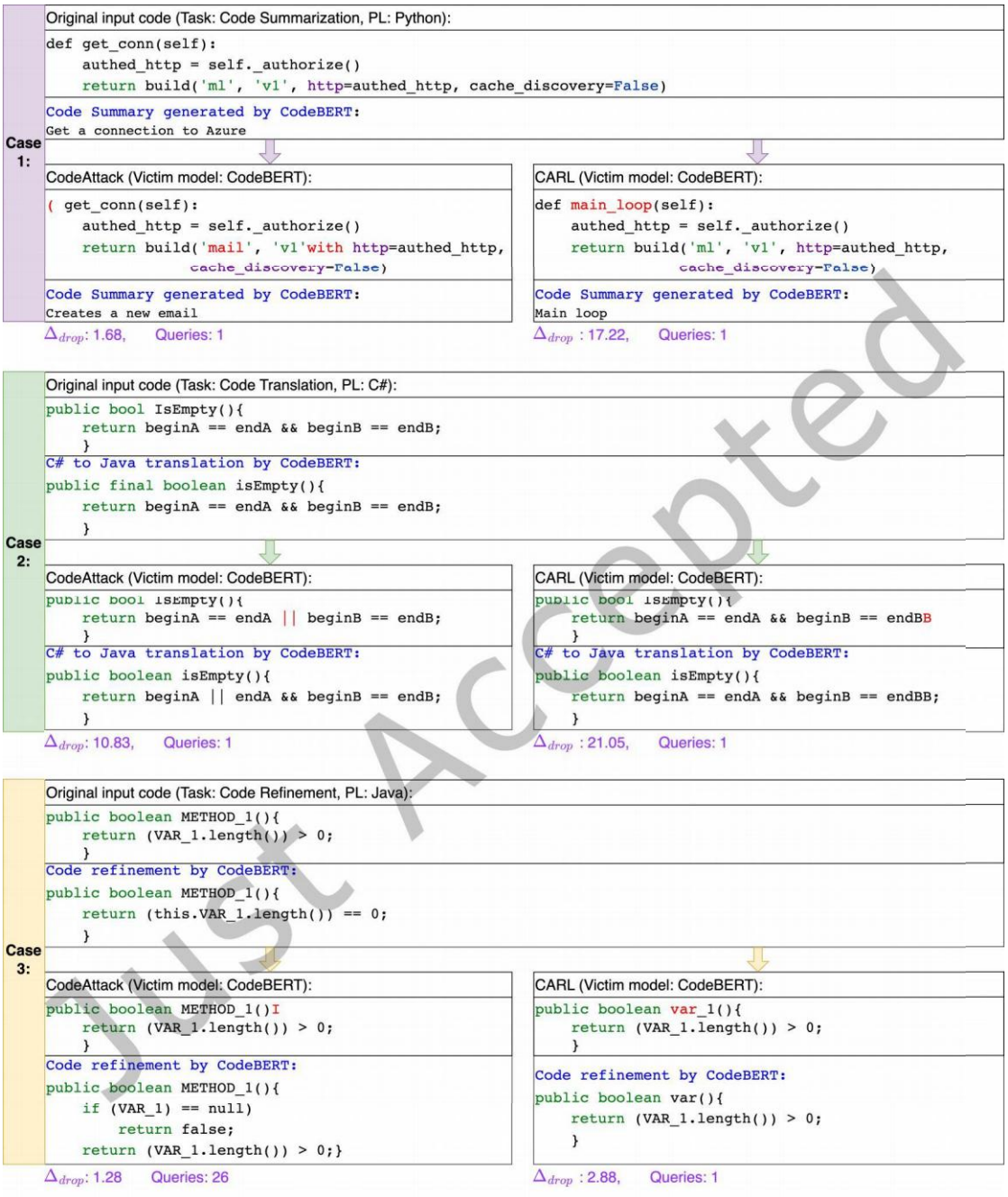

Fig. 6. The case study on code summarizaiton (the first row), code translation (the second row) and code refinement (the third row) tasks. The red texts denote the attack perturbations produced by the attack models.

action sequence sampling, increasing the sampling frequency enhances the discovery of high-quality adversarial samples. However, it comes at the cost of increased time spent computing rewards for the samples, resulting in fewer model updates within a limited training timeframe. We conduct tests with various sampling sizes (1, 5, 10, 50, 100) per step and selected the optimal value based on the average reward observed on the validation set. The number of samples per step directly impacts the trade-off between exploration and exploitation within the action space, thereby influencing the overall performance of our CARL.

External validity: In our experiments, we examine the performance of four widely used pre-trained code models across three generative tasks. It's important to note that we do not delve into classification tasks such as clone detection and vulnerability prediction in our study. While our proposed approach has the potential to extend to other pre-trained models and downstream tasks, its performance may vary. Additionally, we concentrate solely on four prevalent programming languages - Java, Python, C#, and PHP. It's essential to acknowledge that the effectiveness of CARL may differ when applied to other programming languages, such as Ruby and Javascript. Lastly, our CARL is initialized with pre-trained PLM parameters, and it's worth noting that different initializations can impact the convergence results.

9 CONCLUSION

In this paper, we introduced an innovative unsupervised code-based adversarial attack method called CARL. Distinguished from previous greedy or genetic-based approaches, CARL was meticulously designed with learnable parameters, employing reinforcement learning for training and predicting adversarial perturbations to target PLMs. Specifically, our approach leveraged a programming language encoder for semantic representation learning of code and a classification layer for perturbation predictions. We further devised a suite of reward functions to optimize the model parameters using the policy gradient algorithm. Through extensive experiments on code summarization, code translation, and code refinement tasks, involving multiple victim models and programming languages, we demonstrated that CARL achieved the best performance in terms of both attack effectiveness and attack quality.

In the future, we plan to delve deeper into code-based adversarial attack and defense strategies. We intend to enhance the robustness of PLMs in various software engineering tasks by constructing an end-to-end framework for adversarial attack and defense. In addition, the popular large language models like CodeLlama are based on a Transformer decoder architecture. We also plan to explore the use of a generative framework based on large language models to directly generate controllable adversarial examples.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their constructive suggestions. This work is supported by the Postdoctoral Fellowship Program of CPSF (No. GZC20232811), the China Postdoctoral Science Foundation (No. 2024M753357 and No. 2024M750154) and the Key Research Program of Frontier Sciences, CAS (No. ZDBSLY- JSC038).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, 2655–2668.

[2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[3] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani B. Srivastava, and Kai-Wei Chang. 2018. Generating Natural Language Adversarial Examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels,*

Belgium, October 31 - November 4, 2018, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, 2890–2896.

[4] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing robustness of ML-based program analysis tools using metamorphic program transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1377–1381.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186.

[7] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified Language Model Pre-training for Natural Language Understanding and Generation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 13042–13054.

[8] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2021. Glm: General language model pretraining with autoregressive blank infilling. *arXiv preprint arXiv:2103.10360* (2021).

[9] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: White-Box Adversarial Examples for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 2: Short Papers*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, 31–36.

[10] Chuyu Fang, Chuan Qin, Qi Zhang, Kaichun Yao, Jingshuai Zhang, Hengshu Zhu, Fuzhen Zhuang, and Hui Xiong. 2023. RecruitPro: A Pretrained Language Model with Skill-Aware Prompt Learning for Intelligent Recruitment. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, Ambuj K. Singh, Yizhou Sun, Leman Akoglu, Dimitrios Gunopulos, Xifeng Yan, Ravi Kumar, Fatma Ozcan, and Jieping Ye (Eds.). ACM, 3991–4002.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547.

[12] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[13] Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. 2018. Black-Box Generation of Adversarial Text Sequences to Evade Deep Learning Classifiers. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*. IEEE Computer Society, 50–56.

[14] Siddhant Garg and Goutham Ramakrishnan. 2020. BAE: BERT-based Adversarial Examples for Text Classification. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 6174–6181.

[15] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. [n. d.]. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). 7212–7225.

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

[17] Jordan Henke, Goutham Ramakrishnan, Zi Wang, Aws Albarghouth, Somesh Jha, and Thomas Reps. 2022. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 526–537.

[18] Yu-Lun Hsieh, Minhao Cheng, Da-Cheng Juan, Wei Wei, Wen-Lian Hsu, and Cho-Jui Hsieh. 2019. On the robustness of self-attentive models. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 1520–1529.

[19] Hamel Husain, Ho Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). http://arxiv.org/abs/1909.09436

[20] Akshita Jha and Chandan K Reddy. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 14892–14900.

[21] Feihu Jiang, Chuan Qin, Kaichun Yao, Chuyu Fang, Fuzhen Zhuang, Hengshu Zhu, and Hui Xiong. 2024. Enhancing Question Answering for Enterprise Knowledge Bases using Large Language Models. *CoRR* abs/2404.08695 (2024).

[22] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A tree-based pre-trained model for programming language. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021, Virtual Event, 27-30 July 2021*

(Proceedings of Machine Learning Research, Vol. 161), Cassio P. de Campos, Marloes H. Maathuis, and Erik Quaeghebeur (Eds.). AUAI Press, 54–63.

[23] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2020. Is bert really robust? a strong baseline for natural language attack on text classification and entailment. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 8018–8025.

[24] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2020. Is BERT Really Robust? A Strong Baseline for Natural Language Attack on Text Classification and Entailment. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 8018–8025.

[25] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Pre-trained Contextual Embedding of Source Code. *CoRR* abs/2001.00059 (2020).

[26] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 7871–7880.

[27] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. 2019. TextBugger: Generating Adversarial Text Against Real-world Applications. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[28] Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. 2020. BERT-ATTACK: Adversarial Attack Against BERT Using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 6193–6202.

[29] Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. 2020. BERT-ATTACK: Adversarial Attack Against BERT Using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 6193–6202.

[30] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[32] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.).

[33] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[34] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics* 21, 6 (1953), 1087–1092.

[35] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[36] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[37] Danish Pruthi, Bhuwan Dhingra, and Zachary C. Lipton. 2019. Combating Adversarial Misspellings with Robust Word Recognition. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 5582–5591.

[38] Chuan Qin, Kaichun Yao, Hengshu Zhu, Tong Xu, Dazhong Shen, Enhong Chen, and Hui Xiong. 2023. Towards Automatic Job Description Generation With Capability-Aware Neural Networks. *IEEE Trans. Knowl. Data Eng.* 35, 5 (2023), 5341–5355.

[39] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.

[41] Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. 2019. Generating Natural Language Adversarial Examples through Probability Weighted Word Saliency. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 1085–1097.

[42] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020). https://arxiv.org/abs/2009.10297

[43] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[44] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems* 12 (1999).

[45] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1433–1443.

[46] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[47] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[48] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556* (2021).

[49] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. [n. d.]. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). 8696–8708.

[50] Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.* 8 (1992), 229–256.

[51] Puyudi Yang, Jianbo Chen, Cho-Jui Hsieh, Jane-Ling Wang, and Michael I Jordan. 2020. Greedy attack and gumbel attack: Generating adversarial examples for discrete data. *The Journal of Machine Learning Research* 21, 1 (2020), 1613–1648.

[52] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-trained Models of Code. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1482–1493.

[53] Kaichun Yao, Chuan Qin, Hengshu Zhu, Chao Ma, Jingshuai Zhang, Yi Du, and Hui Xiong. 2021. An Interactive Neural Network Approach to Keyphrase Extraction in Talent Recruitment. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong (Eds.). ACM, 2383–2393.

[54] Kaichun Yao, Jingshuai Zhang, Chuan Qin, Xin Song, Peng Wang, Hengshu Zhu, and Hui Xiong. 2023. ResuFormer: Semantic Structure Understanding for Resumes via Multi-Modal Pre-training. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 3154–3167.

[55] Kaichun Yao, Libo Zhang, Tiejian Luo, and Yanjun Wu. 2018. Deep reinforcement learning for extractive document summarization. *Neurocomputing* 284 (2018), 52–62.

[56] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[57] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: continual pre-training on sketches for library-oriented code generation. *arXiv preprint arXiv:2206.06888* (2022).

[58] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards robustness of deep program processing models—detection, estimation, and enhancement. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–40.

[59] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1169–1176.

[60] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568* (2023).

[61] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. 2022. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–30.