CodeAttack: Code-based Adversarial Attacks for Pre-Trained Programming Language Models

Akshita Jha and Chandan K. Reddy

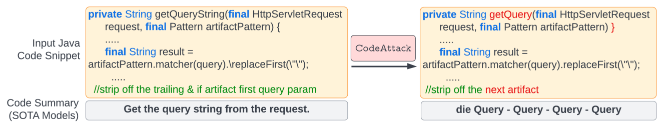
Department of Computer Science, Virginia Tech, Arlington VA - 22203.
akshitajha@vt.edu, reddy@cs.vt.edu

Abstract

Pre-trained programming language (PL) models (such as CodeT5, CodeBERT, GraphCodeBERT, etc.,) have the potential to automate software engineering tasks involving code understanding and code generation. However, these models operate in the natural channel of code, *i.e.*, they are primarily concerned with the human understanding of the code. They are not robust to changes in the input and thus, are potentially susceptible to adversarial attacks in the natural channel. We propose, **CodeAttack**, a simple yet effective black-box attack model that uses code structure to generate effective, efficient, and imperceptible adversarial code samples and demonstrates the vulnerabilities of the state-of-the-art PL models to code-specific adversarial attacks. We evaluate the transferability of CodeAttack on several code-code (translation and repair) and code-NL (summarization) tasks across different programming languages. CodeAttack outperforms state-of-the-art adversarial NLP attack models to achieve the best overall drop in performance while being more efficient, imperceptible, consistent, and fluent. The code can be found at https://github.com/reddy-lab-code-research/CodeAttack.

1 Introduction

There has been a recent surge in the development of general purpose programming language (PL) models (Ahmad et al. 2021; Feng et al. 2020; Guo et al. 2020; Tipirneni, Zhu, and Reddy 2022; Wang et al. 2021). They can capture the relationship between natural language and source code, and potentially automate software engineering development tasks involving code understanding (clone detection, defect detection) and code generation (code-code translation, code-code refinement, code-NL summarization). However, the data-driven pre-training of the above models on massive amounts of code data constraints them to primarily operate in the 'natural channel' of code (Chakraborty et al. 2022; Hindle et al. 2016; Zhang et al. 2022). *This 'natural channel' focuses on conveying information to humans through code comments, meaningful variable names, and function names* (Casalnuovo et al. 2020). In such a scenario, the robustness and vulnerabilities of the pre-trained models need careful investigation. In this work, we leverage the code structure to *generate adversarial samples in the natural channel of code* and demonstrate the vulnerability of the state-of-the-art programming language models to adversarial attacks.



```
private String getQueryString(final HttpServletRequest
    request, final Pattern artifactPattern) {
    .....
    final String result =
artifactPattern.matcher(query).\replaceFirst(\"\");
    .....
  //strip off the trailing & if artifact first query param
```

Get the query string from the request.

```
private String getQuery(final HttpServletRequest
    request, final Pattern artifactPattern) }
    .....
    final String result =
artifactPattern.matcher(query).replaceFirst(\"\");
    .....
 //strip off the next artifact
```

die Query - Query - Query - Query

Figure 1: CodeAttack makes a small modification to the input code snippet (red) which causes significant changes to the code summary obtained from the SOTA pre-trained programming language models. Keywords are highlighted in blue and comments in green.

Adversarial attacks are characterized by imperceptible changes in the input that result in incorrect predictions from a machine learning model. For pre-trained PL models operating in the natural channel, such attacks are important for two primary reasons: (i) *Exposing system vulnerabilities* and *evaluating model robustness*: For example, a small change in the input programming language (akin to a typo or a spelling mistake in the NL scenario) can trigger the code summarization model to generate a gibberish natural language code summary (see Figure 1), and (ii) *Model interpretability*: For example, adversarial samples can be used to inspect the tokens pre-trained PL models attend to.

A successful adversarial attack in the natural channel for code should have the following properties: (i) *Minimal perturbations*: Akin to spelling mistakes or synonym replacement in NL that mislead neural models with imperceptible changes, (ii) *Code Consistency*: Perturbed code is consistent with the original input and follows the same coding style as the original code, and (iii) *Code fluency*: Does not alter the user-level code understanding of the original code. The current natural language adversarial attack models fall short on all three fronts. Hence, we propose **CodeAttack** – a simple yet effective black-box attack model for generating adversarial samples in the natural channel for any input code snippet, irrespective of the programming language.

CodeAttack operates in a realistic scenario, where the adversary does **not** have access to model parameters but only to the test queries and the model prediction. CodeAttack uses a pre-trained masked CodeBERT model (Feng et al. 2020) as the adversarial code generator to generate imperceptible and effective adversarial examples by leveraging the code structure. Our primary contributions are as follows:

- To the best of our knowledge, our work is the first one to detect the vulnerabilities of pre-trained programming language models to adversarial attacks in the natural channel of code. We propose a simple yet effective realistic black-box attack method, CodeAttack, that generates adversarial samples for a code snippet irrespective of the input programming language.
- We design a general purpose black-box attack method for sequence-to-sequence PL models that is transferable across different downstream tasks like code translation, repair, and summarization. The input language agnostic nature of our method also makes it extensible to sequence-to-sequence tasks in other domains.
- We demonstrate the effectiveness of CodeAttack over existing NLP adversarial models through an extensive empirical evaluation. CodeAttack outperforms the natural language baselines when considering both the attack quality and its efficacy.

2 Background and Related Work

Dual Channel of Source Code. Casalnuovo et al. (2020) proposed a dual channel view of code: (i) formal, and (ii) natural. The formal channel is precise and used for code execution by compilers and interpreters. The natural language channel, on the other hand, is for human comprehension and is noisy. It relies on code comments, variable names, function names, etc., to ease human understanding. The state-of-the-art PL models operate primarily in the natural channel of code (Zhang et al. 2022) and therefore, we generate *adversarial samples by making use of this natural channel.*

Adversarial Attacks in NLP. BERT-Attack (Li et al. 2020) and BAE (Garg and Ramakrishnan 2020) use BERT for attacking vulnerable words. TextFooler (Jin et al. 2020) and PWWS (Ren et al. 2019) use synonyms and part-of-speech (POS) tagging to replace important tokens. Deepwordbug (Gao et al. 2018) and TextBugger (Li et al. 2019) use character insertion, deletion, and replacement strategy for attacks whereas Hsieh et al. (2019) and Yang et al. (2020) use a greedy search and replacement strategy. Alzantot et al. (2018) use genetic algorithm and Ebrahimi et al. (2018), Papernot et al. (2016), and Pruthi, Dhingra, and Lipton (2019) use model gradients for finding subsititutes. None of these methods have been designed specifically for programming languages, which is more structured than natural language.

Adversarial Attacks for PL. Zhang et al. (2020) generate adversarial examples by renaming identifiers using Metropolis-Hastings sampling (Metropolis et al. 1953). Yang et al. (2022) improve on that by using greedy and genetic algorithm. Yefet, Alon, and Yahav (2020) use gradient based exploration; whereas Applis, Panichella, and van Deursen (2021) and (Henkel et al. 2022) propose metamorphic transformations for attacks. The above models focus on classification tasks like defect detection and clone detection. Although some works do focus on adversarial examples for code summarization (Henkel et al. 2022; Zhou et al. 2022), they do not do so in the natural channel. They also do not test the transferability to different tasks, PL models, and different programming languages. Our model, CodeAttack, assumes black-box access to the state-of-the-art PL models for generating adversarial attacks for code generation tasks like code translation, code repair, and code summarization using a constrained code-specific greedy algorithm to find meaningful substitutes for vulnerable tokens, irrespective of the input programming language.

3 CodeAttack

We describe the capabilities, knowledge, and the goal of the proposed model, and provide details on how it detects vulnerabilities in the state-of-the-art pre-trained PL models.

Threat Model

Adversary's Capabilities. The adversary is capable of perturbing the test queries given as input to a pre-trained PL model to generate adversarial samples. We follow the existing literature for generating natural language adversarial examples and allow for two types of perturbations for the input code sequence in the natural channel: (i) character-level perturbations, and (ii) token-level perturbations. The adversary is allowed to perturb only a certain number of tokens/characters and must ensure a high similarity between the original code and the perturbed code. Formally, for a given input code sequence $\mathcal{X} \in X$, where X is the input space, a valid adversarial code example $\mathcal{X}_{adv}$ satisfies the requirements:

$$\mathcal{X} \neq \mathcal{X}_{adv} \tag{1}$$

$$\mathcal{X}_{adv} \leftarrow \mathcal{X} + \delta; \qquad \text{s.t. } ||\delta|| < \theta \tag{2}$$

$$\text{Sim}(\mathcal{X}_{adv}, \mathcal{X}) \geq \epsilon \tag{3}$$

where θ is the maximum allowed adversarial perturbation; $\text{Sim}(\cdot)$ is a similarity function; and ϵ is the similarity threshold. We describe the perturbation constraints and the similarity functions in more detail in Section 3.

Adversary's Knowledge. We assume standard black-box access to realistically assess the vulnerabilities and robustness of existing pre-trained PL models. The adversary does *not* have access to the model parameters, model architecture, model gradients, training data, or the loss function. It can only query the pre-trained PL model with input sequences and get their corresponding output probabilities. This is more practical than a white-box scenario where the attacker assumes access to all the above.

Adversary's Goal. Given an input code sequence as query, the adversary's goal is to degrade the quality of the generated output sequence through imperceptibly modifying the query in the natural channel of code. The generated output sequence can either be a code snippet (code translation, code repair) or natural language text (code summarization). Formally, given a pre-trained PL model $F : X \rightarrow Y$, where X is the input space, and Y is the output space, the goal of the adversary is to generate an adversarial sample $\mathcal{X}_{adv}$ for an input sequence $\mathcal{X}$ *s.t.*

$$F(\mathcal{X}_{adv}) \neq F(\mathcal{X}) \tag{4}$$

$$Q(F(\mathcal{X})) - Q(F(\mathcal{X}_{adv})) \geq \phi \tag{5}$$

where $Q(\cdot)$ measures the quality of the generated output and ϕ is the specified drop in quality. This is in addition to the constraints applied on $\mathcal{X}_{adv}$ earlier. We formulate our final problem of generating adversarial samples as follows:

$$\Delta_{atk} = \text{argmax}_{\delta} \ [Q(F(\mathcal{X})) - Q(F(\mathcal{X}_{adv}))] \tag{6}$$

In the above objective function, $\mathcal{X}_{adv}$ is a minimally perturbed adversary subject to constraints on the perturbations δ (Eqs.1-5). CodeAttack searches for a perturbation Δ_{atk} to maximize the difference in the quality $Q(\cdot)$ of the output sequence generated from the original input code snippet $\mathcal{X}$ and that by the perturbed code snippet $\mathcal{X}_{adv}$.

Attack Methodology

There are two primary steps: (i) Finding the most vulnerable tokens, and (ii) Substituting these vulnerable tokens (subject to code-specific constraints), to generate adversarial samples in the natural channel of code.

Finding Vulnerable Tokens CodeBERT gives more attention to keywords and identifiers while making predictions (Zhang et al. 2022). We leverage this information and hypothesize that certain input tokens contribute more towards the final prediction than others. 'Attacking' these highly influential or highly vulnerable tokens increases the probability of altering the model predictions more significantly as opposed to attacking non-vulnerable tokens. Under a black-box setting, the model gradients are unavailable and the adversary only has access to the output logits of the pre-trained PL model. We define 'vulnerable tokens' as tokens having a high influence on the output logits of the model. Let F be an encoder-decoder pre-trained PL model. The given input sequence is denoted by $\mathcal{X} = [x_1, .., x_i, ..., x_m]$, where $\{x_i\}_1^m$ are the input tokens. The output is a sequence of vectors: $\mathcal{O} = F(\mathcal{X}) = [o_1, ..., o_n]$; $y_t = \text{argmax}(o_t)$; where $\{o_t\}_1^n$ is the output logit for the correct output token y_t for the time step t. Without loss of generality, we can also assume the output sequence $\mathcal{Y} = F(\mathcal{X}) = [y_i, ..., y_l]$. $\mathcal{Y}$ can either be a sequence of code or natural language tokens.

To find the vulnerable input tokens, we replace a token x_i with [MASK] such that $\mathcal{X}_{\setminus x_i} = [x_1, ., x_{i-1}, \text{[MASK]}, x_{i+1}, ., x_m]$ and get its output logits. The output vectors are now $\mathcal{O}_{\setminus x_i} = F(\mathcal{X}_{\setminus x_i}) = [o'_1, ..., o'_q]$ where $\{o'_t\}_1^q$ is the new output logit for the correct prediction $\mathcal{Y}$. The influence score for the token x_i is as follows:

$$I_{x_i} = \sum_{t=1}^{n} o_t - \sum_{t=1}^{q} o'_t \tag{7}$$

We rank all the tokens according to their influence score I_{x_i} in descending order to find the most vulnerable tokens V. We select the top-k tokens to limit the number of perturbations and attack them iteratively either by replacing them or by inserting/deleting a character around them.

Token Class	Description
Keywords	Reserved word
Identifiers	Variable, Class Name, Method name
Operators	Brackets ({},(),[]), Symbols (+,*,/,-,%,;,.)
Arguments	Integer, Floating point, String, Character

Table 1: Token class and their description.

Substituting Vulnerable Tokens We adopt greedy search using a masked programming language model, subject to code-specific constraints, to find substitutes S for vulnerable tokens V such that they are minimally perturbed and have the maximal probability of incorrect prediction.

Search Method. In a given input sequence, we mask a vulnerable token v_i and use the masked PL model to predict a meaningful contextualized token in its place. We use the top-k predictions for each of the masked vulnerable tokens as our initial search space. Let $\mathcal{M}$ denote a masked PL model. Given an input sequence $\mathcal{X} = [x_1, .., v_i, .., x_m]$, where v_i is a vulnerable token, $\mathcal{M}$ uses WordPiece algorithm (Wu et al. 2016) for tokenization that breaks uncommon words into sub-words resulting in $\mathcal{H} = [h_1, h_2, .., h_q]$. We align and mask all the corresponding sub-words for v_i, and combine the predictions to get the top-k substitutes $S' = \mathcal{M}(H)$ for the vulnerable token v_i. This initial search space S' consists of l possible substitutes for a vulnerable token v_i. We then filter out substitute tokens to ensure minimal perturbation, code consistency, and code fluency of the generated adversarial samples, subject to code-specific constraints.

Code-Specific Constraints. Since the tokens generated from a masked PL model may not be meaningful individual code tokens, we further use a CodeNet tokenizer (Puri et al. 2021) to break a token into its corresponding code tokens. The code tokens are tokenized into four primary code token classes (Table 1). If s_i is the substitute for the vulnerable token v_i as tokenized by $\mathcal{M}$, and $Op(\cdot)$ denotes the operators present in any given token using CodeNet tokenizer, we allow the substitute tokens to have an extra or a missing operator (akin to typos in the natural channel of code).

$$|Op(v_i)| - 1 \leq |Op(s_i)| \leq |Op(v_i)| + 1 \tag{8}$$

Let $C(\cdot)$ denote the code token class (identifiers, keywords, and arguments) of a token. We maintain the alignment between between v_i and the potential substitute s_i as follows.

$$C(v_i) = C(s_i) \text{ and } |C(v_i)| = |C(s_i)| \tag{9}$$

The above code constraints maintain the code fluency and the code consistency of $\mathcal{X}_{adv}$ and significantly reduce the search space for finding adversarial examples.

Substitutions. We allow two types of substitutions of vulnerable tokens to generate adversarial examples: (i) *Operator (character) level substitution* – only an operator is inserted/replaced/deleted; and (ii) *Token-level substitution*. We use the reduced search space S and iteratively substitute, until the adversary's goal is met. We only allow replacing upto $p\%$ of the vulnerable tokens/characters to limit the number of perturbations. We also maintain the cosine similarity between the input text $\mathcal{X}$ and the adversarially perturbed text $\mathcal{X}_{adv}$ above a certain threshold (Equation 3). The complete algorithm is given in Algorithm 1. CodeAttack maintains minimal perturbation, code fluency, and code consistency between the input and the adversarial code snippet.

Algorithm 1: CodeAttack: Generating adversarial examples for Code

```
Input: Code $\mathcal{X}$; Victim model $F$; Maximum perturbation $\theta$; Similarity $\epsilon$; Performance Drop $\phi$
Output: Adversarial Example $\mathcal{X}_{adv}$
Initialize: $\mathcal{X}_{adv} \leftarrow \mathcal{X}$
//Find vulnerable tokens 'V'
for $x_i$ in $\mathcal{M}(\mathcal{X})$ do
    Calculate $I_{x_i}$ acc. to Eq.(7)
end
$V \leftarrow$ Rank$(x_i)$ based on $I_{x_i}$
//Find substitutes 'S'
for $v_i$ in $V$ do
    $S \leftarrow$ Filter$(v_i)$ subject to Eqs.(8), (9)
    for $s_j$ in $S$ do
        //Attack the victim model
        $\mathcal{X}_{adv} = [x_1, ..., x_{i-1}, s_j, ..., x_m]$
        if $Q(F(\mathcal{X})) - Q(F(\mathcal{X}_{adv})) \geq \phi$ and $Sim(\mathcal{X}, \mathcal{X}_{adv}) \geq \epsilon$ and $||\mathcal{X}_{adv} - \mathcal{X}|| \leq \theta$ then
            return $\mathcal{X}_{adv}$ //Success
        end
    end
    //One perturbation
    $\mathcal{X}_{adv} \leftarrow [x_1, ...x_{i-1}, s_j, ..x_m]$
end
return
```

4 Experiments

We study the following research questions:

- **RQ1:** How effective and transferable are the attacks generated using CodeAttack to different downstream tasks and programming languages?
- **RQ2:** How is the quality of adversarial samples generated using CodeAttack?
- **RQ3:** Is CodeAttack effective when we limit the number of allowed perturbations?
- **RQ4:** What is the impact of different components on the performance of CodeAttack?

Downstream Tasks and Datasets We evaluate the transferability of CodeAttack across different sequence to sequence downstream tasks and in different programming languages: (i) *Code Translation*[1] involves translating between C# and Java and vice-versa, (ii) *Code Repair* automatically fixes bugs in Java functions. We use the 'small' dataset (Tufano et al. 2019), (iii) *Code Summarization* involves generating natural language summary for a given code. We use Python, Java, and PHP from the CodeSearchNet dataset (Husain et al. 2019). (See Appendix A for details).

[1]https://github.com/eclipse/jgit/, http://lucene.apache.org/, http://poi.apache.org/, https://github.com/antlr/

Victim Models We pick a representative method from different categories for our experiments: (i) *CodeT5*: Pre-trained encoder-decoder transformer-based PL model (Wang et al. 2021), (ii) *CodeBERT*: Bimodal pre-trained PL model (Feng et al. 2020), (iii) *GraphCodeBERT*: Pre-trained graph PL model (Guo et al. 2020), (iv) *RoBERTa*: Pre-trained NL model (Guo et al. 2020). (See Appendix A for details).

Baseline Models Since CodeAttack operates in the natural channel of code, we compare with two state-of-the-art adversarial NLP baselines for a fair comparison: (i) TextFooler: Uses synonyms, Part-Of-Speech checking, and semantic similarity to generate adversarial text (Jin et al. 2020), (ii) BERT-Attack: Uses a pre-trained BERT masked language model to generate adversarial text (Li et al. 2020).

Evaluation Metrics We evaluate the *effectiveness* and the *quality* of the generated adversarial code.

Attack Effectiveness. To measure the effectiveness of the adversarial attacks on sequence-to-sequence tasks, we define the following metric.

- Δ_{drop}**:** We measure the drop in the downstream performance *before* and *after* the attack using CodeBLEU (Ren et al. 2020) and BLEU (Papineni et al. 2002). We define

$$\Delta_{drop} = Q_{\text{before}} - Q_{\text{after}} = Q(F(\mathcal{X}), \mathcal{Y}) - Q(F(\mathcal{X}_{adv}, \mathcal{Y})$$

  where $Q = \{\text{CodeBLEU, BLEU}\}$; $\mathcal{Y}$ is the ground truth output; F is the pre-trained victim PL model, $\mathcal{X}_{adv}$ is the adversarial code sequence generated after perturbing the original input source code $\mathcal{X}$. CodeBLEU measures the quality of the *generated* code snippet for code translation and code repair, and BLEU measures the quality of the *generated* natural language code summary when compared to the ground truth.
- **Success %:** Computes the % of successful attacks as measured by Δ_{drop}. The higher the value, the more *effective* is the adversarial attack.

Attack Quality. The following metric measures the quality of the generated adversarial code across three dimensions: (i) efficiency, (ii) imperceptibility, and (iii) code consistency.

- **# Queries:** Under a black-box setting, the adversary can query the victim model to check for changes in the output logits. The lower the average number of queries required per sample, the more *efficient* is the adversary.
- **# Perturbation:** The number of tokens changed on an average to generate an adversarial code. The lower the value, the more *imperceptible* the attack will be.
- **CodeBLEU$_q$:** Measures the consistency of the adversarial code using **CodeBLEU**$_q$ = CodeBLEU$(\mathcal{X}, \mathcal{X}_{adv})$; where $\mathcal{X}_{adv}$ is the adversarial code sequence generated after perturbing the original input source code $\mathcal{X}$. The higher the CodeBLEU$_q$, the more *consistent* the adversarial code is with the original source code.

Task	Victim Model	Attack Method	Attack Effectiveness				Attack Quality		
			Before	After	Δ_{drop}	Success%	#Queries	#Perturb	CodeBLEU$_q$
Translate (Code-Code)	CodeT5	TextFooler	73.99	68.08	5.91	28.29	94.95	2.90	63.19
		BERT-Attack		63.01	10.98	75.83	163.5	5.28	62.52
		CodeAttack		**61.72**	**12.27**	**89.3**	**36.84**	**2.55**	**65.91**
	CodeBERT	TextFooler	71.16	60.45	10.71	49.2	73.91	1.74	66.61
		BERT-Attack		58.80	12.36	70.1	290.1	5.88	52.14
		CodeAttack		**54.14**	**17.03**	**97.7**	**26.43**	**1.68**	**66.89**
	GraphCode-BERT	Textfooler	66.80	46.51	20.29	38.70	83.17	1.82	63.62
		BERT-Attack		**36.54**	**30.26**	94.33	175.8	6.73	52.07
		CodeAttack		38.81	27.99	**98**	**20.60**	**1.64**	**65.39**
Repair (Code-Code)	CodeT5	Textfooler	61.13	57.59	3.53	58.84	90.50	2.36	**69.53**
		BERT-Attack		**52.70**	**8.43**	94.33	262.5	15.1	53.60
		CodeAttack		53.21	7.92	**99.36**	**30.68**	**2.11**	69.03
	CodeBERT	Textfooler	61.33	53.55	7.78	81.61	45.89	2.16	**68.16**
		BERT-Attack		**51.95**	**9.38**	95.31	183.3	15.7	61.95
		CodeAttack		52.02	9.31	**99.39**	**25.98**	**1.64**	68.05
	GraphCode-BERT	Textfooler	62.16	54.23	7.92	78.92	51.07	2.20	**67.89**
		BERT-Attack		53.33	8.83	96.20	174.1	15.7	53.66
		CodeAttack		**51.97**	**10.19**	**99.52**	**24.67**	**1.67**	66.16
Summarize (Code-NL)	CodeT5	TextFooler	20.06	14.96	5.70	64.6	410.15	**6.38**	**53.91**
		BERT-Attack		11.96	8.70	78.4	1014.1	7.32	51.34
		CodeAttack		**11.06**	**9.59**	**82.8**	**314.87**	10.1	52.67
	CodeBERT	Textfooler	19.76	14.38	5.37	61.10	358.43	2.92	**54.10**
		BERT-Attack		11.30	8.35	56.47	1912.6	15.8	46.24
		CodeAttack		**10.88**	**8.87**	**88.32**	**204.46**	**2.57**	52.95
	RoBERTa	TextFooler	19.06	14.06	4.99	62.60	356.68	2.80	**54.11**
		BERT-Attack		11.34	7.71	60.46	1742.3	17.1	46.95
		CodeAttack		**10.98**	**8.08**	**87.51**	**183.22**	**2.62**	53.03

Table 2: Results on translation (C#-Java), repair (Java-Java), and summarization (PHP) tasks. The performance is measured in CodeBLEU for Code-Code tasks and in BLEU for Code-NL task. The best result is in **boldface**; the next best is underlined.

Implementation Details The model is implemented in PyTorch. We use the publicly available pre-trained CodeBERT (MLM) masked model as the adversarial code generator. We select the top 50 predictions for each vulnerable token as the initial search space and allow attacking a maximum of 40% of code tokens. The cosine similarity threshold between the original code and adversarially generated code is set to 0.5. As victim models, we use the publicly available fine-tuned checkpoints for CodeT5 and fine-tune CodeBERT, GraphCodeBERT, and RoBERTa on the related downstream tasks. We use a batch-size of 256. All experiments were conducted on a 48 GiB RTX 8000 GPU. The source code for CodeAttack can be found at https://github.com/reddy-lab-code-research/CodeAttack.

RQ1: Effectiveness of CodeAttack

We test the effectiveness and transferability of the generated adversarial samples on three different sequence-to-sequence tasks (Code Translation, Code Repair, and Code Summarization). We generate adversarial code for four different programming languages (C#, Java, Python, and PHP), and attack four different pre-trained PL models (CodeT5, GraphCodeBERT, CodeBERT, and Roberta). The results for C#-Java translation task and for the PHP code summarization task are shown in Table 2. (See Appendix A for Java-C# translation and Python and Java code summarization tasks). CodeAttack has the highest success% compared to other adversarial NLP baselines. CodeAttack also outperforms the adversarial baselines, BERT-Attack and TextFooler, in 6 out of 9 cases – the average Δ_{drop} using CodeAttack is around 20% for code translation and 10% for code repair tasks, respectively. For code summarization, CodeAttack reduces BLEU by almost 50% for all the victim models. As BERT-Attack replaces tokens indiscriminately, its Δ_{drop} is higher in some cases but its attack quality is the lowest.

RQ2: Quality of Attacks using CodeAttack

Quantitative Analysis. Compared to the other adversarial NLP models, CodeAttack is the most efficient as it requires the lowest number of queries for a successful attack (Table 2). CodeAttack is also the least perceptible as the average number of perturbations required are 1-3 tokens in 8 out of 9 cases. The code consistency of adversarial samples, as measured by CodeBLEU$_q$, generated using CodeAttack is comparable to TextFooler which has a very low success rate. CodeAttack has the best overall performance.

Original Code	TextFooler	BERT-Attack	CodeAttack
`public override void WriteByte( byte b) { if (outerInstance.upto == outerInstance.blockSize) {... }}`	`audiences revoked canceling WriteByte(byte b) { if (outerInstance.upto == outerInstance.blockSize) {.... }}`	`public override void ; . b) { if (outerInstance.upto == outerInstance.blockSize) {... }}`	`public override void WriteByte( bytes b) { if (outerInstance.upto == outerInstance.blockSize) {... }}`
CodeBLEU$_{\text{before}}$:100	Δ_{drop}:5.74; CodeBLEU$_q$: 63.28	Δ_{drop}:27.26; CodeBLEU$_q$:49.87	Δ_{drop}:20.04; CodeBLEU$_q$: 91.69

Table 3: Qualitative examples of adversarial codes on C#-Java Code Translation task. (See Appendix A for more examples).

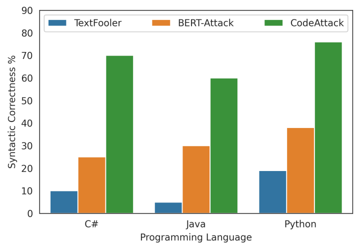


Figure 2: Syntactic correctness of adversarial code on C#, Java, and Python demonstrating attack quality.

Qualitative Analysis. Table 3 presents qualitative examples of the generated adversarial code snippets from different attack models. Although TextFooler has a slightly better CodeBLEU$_q$ score when compared to CodeAttack (as seen from Table 2), it replaces keywords with closely related natural language words (`public` $\rightarrow$ `audiences`; `override` $\rightarrow$ `revoked`, `void` $\rightarrow$ `cancelling`). BERT-Attack has the lowest CodeBLEU$_q$ and substitutes tokens with seemingly random words. Both TextFooler and BERT-Attack have not been designed for programming languages. CodeAttack generates more meaningful adversarial code samples by replacing vulnerable tokens with variables and operators which are imperceptible and consistent.

Syntactic correctness. Syntactic correctness of the generated adversarial code is a useful criteria for evaluating the attack quality even though CodeAttack and other PL models primarily operate in the natural channel of code, *i.e.*, they are concerned with code understanding for humans and not with the execution or compilation of the code. The datasets described earlier consist of code snippets and cannot be compiled. Therefore, we generate adversarial code for C#, Java, and Python using TextFooler, BERT-Attack, and CodeAttack and ask 3 human annotators, familiar with these languages to verify the syntax manually. We randomly sample 60 generated adversarial codes for all three programming languages for evaluating each of the above methods. CodeAttack has the highest average syntactic correctness for C# (70%), Java (60%), and Python (76.19%) followed by BERT-Attack and TextFooler (Figure 2), further highlighting the need for a code-specific adversarial attack.

RQ3: Limiting Perturbations using CodeAttack

We restrict the number of perturbations when attacking a pre-trained PL model to a strict limit, and study the effectiveness of CodeAttack. (See Figure 3). From Figure 3a, we observe that, as the perturbation % increases, the CodeBLEU$_{\text{after}}$ for CodeAttack decreases but remains constant for TextFooler and only slightly decreases for BERT-Attack. We also observe that, although CodeBLEU$_q$ for CodeAttack is the second best (Figure 3b), it has the highest attack success rate (Figure 3d) and requires the lowest number of queries for a successful attack (Figure 3c). This shows the efficiency of CodeAttack and the need for code-specific adversarial attacks.

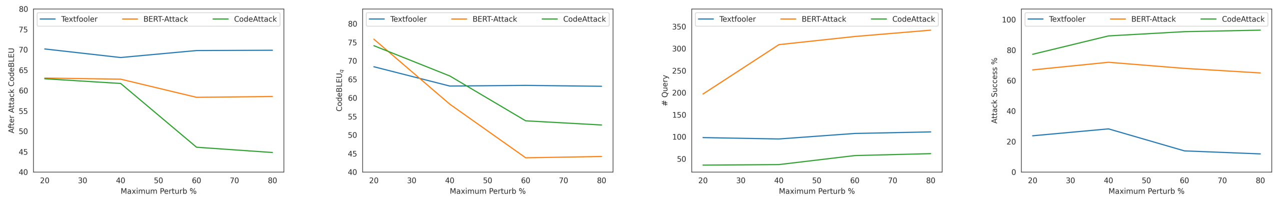
Ablation Study

Importance of Vulnerable Tokens. We create a variant, CodeAttack$_{\texttt{RAND}}$, which randomly samples tokens from the input code for substitution. We define another variant, CodeAttack$_{\texttt{VUL}}$, which finds vulnerable tokens based on logit information (Section 3) and attacks them, albeit without any constraints. As can be seen from Figure 4a, attacking random tokens is not as effective as attacking vulnerable tokens. Using CodeAttack$_{\texttt{VUL}}$ yields greater Δ_{drop} and requires fewer number of queries when compared to CodeAttack$_{\texttt{RAND}}$, across all three models at similar CodeBLEU$_q$ (Figure 4b) and success % (Figure 4d).

Importance of Code-Specific Constraints. We find vulnerable tokens and apply two types of constraints: (i) Operator level constraint (CodeAttack$_{\texttt{OP}}$), and (ii) Token level constraint (CodeAttack$_{\texttt{TOK}}$) (Section 3). Only applying the operator level constraint results in lower attack success% (Figure 4d) and a lower Δ_{drop} (Figure 4a) but a much higher CodeBLEU$_q$. This is because we limit the changes only to operators resulting in minimal changes. On applying both operator level and token level constraints together, the Δ_{drop} and the attack success% improve significantly. (See Appendix A for qualitative examples.)

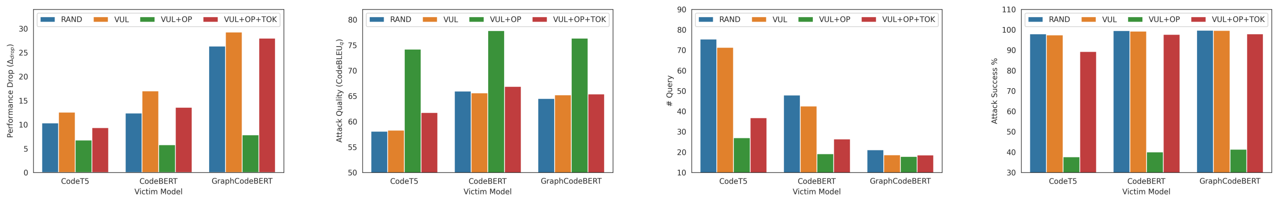
Overall, the final model, CodeAttack, which consists of the components CodeAttack$_{\texttt{VUL}}$, CodeAttack$_{\texttt{OP}}$, and CodeAttack$_{\texttt{TOK}}$ has the best trade-off across Δ_{drop}, attack success %, CodeBLEU$_q$, and # Queries for all three pre-trained PL victim models.

Human Evaluation. We sample 50 original and perturbed Java and C# code samples and shuffle them to create a mix. We ask 3 human annotators, familiar with the two programming languages, to classify the codes as either original or adversarial by evaluating the source codes in their natural channel. On an average, 72.1% of the given codes were classified as original. We also ask them to read the given adversarial codes and rate their code understanding on a scale of 1 to 5; where 1 corresponds to 'Code cannot be understood at all'; and 5 corresponds to 'Code is completely understandable'. The average code understanding for the adversarial codes was 4.14. Additionally, we provide the an-



(a) CodeBLEU$_{\text{after}}$ (b) CodeBLEU$_q$ (c) Average #Queries (d) Attack%

Figure 3: Varying the perturbation % to study attack effectiveness on CodeT5 for the code translation task (C#-Java).



(a) Performance Drop (b) CodeBLEU$_q$ (c) # Queries (d) Average Success Rate

Figure 4: Ablation Study for Code Translation (C#-Java): Performance of different components of CodeAttack with random (RAND) and vulnerable tokens (VUL) and two code-specific constraints: (i) Operator level (OP), and (ii) Token level (TOK).

notators with pairs of adversarial and original codes and ask them to rate the code consistency between the two using a scale between 0 to 1; where 0 corresponds to 'Not at all consistent with the original code', and 1 corresponds to 'Extremely consistent with the original code'. On average, the code consistency was 0.71.

5 Discussion

Humans 'summarize' code by reading function calls, focusing on information denoting the intention of the code (such as variable names) and skimming over structural information (such as `while` and `for` loops) (Rodeghero et al. 2014). Pre-trained PL models operate in a similar manner and do not assign high attention weights to the grammar or the code structure (Zhang et al. 2022). They treat software code as natural language (Hindle et al. 2016) and do not focus on compilation or execution of the input source code before processing them to generate an output (Zhang et al. 2022). Through extensive experimentation, we demonstrate that this limitation of the state-of-the-art PL models can be exploited to generate adversarial examples in the natural channel of code and significantly alter their performance.

We observe that it is easier to attack the code translation task rather than code repair or code summarization tasks. Since code repair aims to fix bugs in the given code snippet, it is more challenging to attack but not impossible. For code summarization, the BLEU score drops by almost 50%. For all three tasks, CodeT5 is comparatively more robust whereas GraphCodeBERT is the most susceptible to attacks using CodeAttack. CodeT5 has been pre-trained on the task of 'Masked Identifier Prediction' or deobsfuction (Lachaux et al. 2021) where changing the identifier names does not have an impact on the code semantics. This helps the model avoid the attacks which involve changing the identifier names. GraphCodeBERT uses data flow graphs in their pre-training which relies on predicting the relationship between the identifiers. Since CodeAttack modifies the identifiers and perturbs the relationship between them, it proves to be extremely effective on GraphCodeBERT. This results in a more significant Δ_{drop} on GraphCodeBERT compared to other models for the code translation task.

The adversarial examples from CodeAttack, although effective in the natural channel of code, can be avoided if the pre-trained PL models compile/execute the code before processing it. This highlights the need to incorporate explicit code structure in the pre-training stage to learn more robust program representations.

6 Conclusion

We introduce, CodeAttack, a black-box adversarial attack model to detect vulnerabilities of the state-of-the-art programming language models. It finds the most vulnerable tokens in a given code snippet and uses a greedy search mechanism to identify contextualized substitutes subject to code-specific constraints. Our model generates adversarial examples in the natural channel of code. We perform an extensive empirical and human evaluation to demonstrate the transferability of CodeAttack on several code-code and code-NL tasks across different programming languages. CodeAttack outperforms the existing state-of-the-art adversarial NLP models, in terms of its attack effectiveness, attack quality, and syntactic correctness. The adversarial samples generated using CodeAttack are efficient, effective, imperceptible, fluent, and code consistent. CodeAttack highlights the need for code-specific adversarial attacks for pre-trained PL models in the natural channel.

References

Ahmad, W.; Chakraborty, S.; Ray, B.; and Chang, K.-W. 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*, 2655–2668. 1

Alzantot, M.; Sharma, Y.; Elgohary, A.; Ho, B.-J.; Srivastava, M.; and Chang, K.-W. 2018. Generating Natural Language Adversarial Examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2890–2896. 2

Applis, L.; Panichella, A.; and van Deursen, A. 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)*, 1377–1381. IEEE. 2

Casalnuovo, C.; Barr, E. T.; Dash, S. K.; Devanbu, P.; and Morgan, E. 2020. A theory of dual channel constraints. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 25–28. IEEE. 1, 2

Chakraborty, S.; Ahmed, T.; Ding, Y.; Devanbu, P. T.; and Ray, B. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 18–30. 1

Ebrahimi, J.; Rao, A.; Lowd, D.; and Dou, D. 2018. HotFlip: White-Box Adversarial Examples for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 31–36. 2

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. 1, 4, 10

Gao, J.; Lanchantin, J.; Soffa, M. L.; and Qi, Y. 2018. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*, 50–56. IEEE. 2

Garg, S.; and Ramakrishnan, G. 2020. BAE: BERT-based Adversarial Examples for Text Classification. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 6174–6181. 2

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Shujie, L.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. 1, 4, 10

Henkel, J.; Ramakrishnan, G.; Wang, Z.; Albarghouthi, A.; Jha, S.; and Reps, T. 2022. Semantic Robustness of Models of Source Code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 526–537. 2

Hindle, A.; Barr, E. T.; Gabel, M.; Su, Z.; and Devanbu, P. 2016. On the naturalness of software. *Communications of the ACM*, 59(5): 122–131. 1, 7

Hsieh, Y.-L.; Cheng, M.; Juan, D.-C.; Wei, W.; Hsu, W.-L.; and Hsieh, C.-J. 2019. On the robustness of self-attentive models. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 1520–1529. 2

Husain, H.; Wu, H.-H.; Gazit, T.; Allamanis, M.; and Brockschmidt, M. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*. 4, 10

Jin, D.; Jin, Z.; Zhou, J. T.; and Szolovits, P. 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*, volume 34, 8018–8025. 2, 4, 10

Lachaux, M.-A.; Roziere, B.; Szafraniec, M.; and Lample, G. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. *Advances in Neural Information Processing Systems*, 34. 7

Lai, G.; Xie, Q.; Liu, H.; Yang, Y.; and Hovy, E. 2017. RACE: Large-scale ReAding Comprehension Dataset From Examinations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 785–794. 10

Li, J.; Ji, S.; Du, T.; Li, B.; and Wang, T. 2019. TextBugger: Generating Adversarial Text Against Real-world Applications. In *26th Annual Network and Distributed System Security Symposium*. 2

Li, L.; Ma, R.; Guo, Q.; Xue, X.; and Qiu, X. 2020. BERT-ATTACK: Adversarial Attack Against BERT Using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 6193–6202. 2, 4, 10

Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; and Stoyanov, V. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*. 10

Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C. B.; Drain, D.; Jiang, D.; Tang, D.; Li, G.; Zhou, L.; Shou, L.; Zhou, L.; Tufano, M.; Gong, M.; Zhou, M.; Duan, N.; Sundaresan, N.; Deng, S. K.; Fu, S.; and Liu, S. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR*, abs/2102.04664. 10

Metropolis, N.; Rosenbluth, A. W.; Rosenbluth, M. N.; Teller, A. H.; and Teller, E. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6): 1087–1092. 2

Papernot, N.; Faghri, F.; Carlini, N.; Goodfellow, I.; Feinman, R.; Kurakin, A.; Xie, C.; Sharma, Y.; Brown, T.; Roy, A.; et al. 2016. Technical report on the cleverhans v2. 1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*. 2

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 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. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics. 4

Pruthi, D.; Dhingra, B.; and Lipton, Z. C. 2019. Combating Adversarial Misspellings with Robust Word Recognition. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 5582–5591. 2

Puri, R.; Kung, D. S.; Janssen, G.; Zhang, W.; Domeniconi, G.; Zolotov, V.; Dolby, J.; Chen, J.; Choudhury, M.; Decker, L.; et al. 2021. Project codenet: a large-scale AI for code dataset for learning a diversity of coding tasks. 3

Rajpurkar, P.; Zhang, J.; Lopyrev, K.; and Liang, P. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2383–2392. Austin, Texas: Association for Computational Linguistics. 10

Ren, S.; Deng, Y.; He, K.; and Che, W. 2019. Generating Natural Language Adversarial Examples through Probability Weighted Word Saliency. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 1085–1097. Florence, Italy: Association for Computational Linguistics. 2

Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; and Ma, S. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*. 4

Rodeghero, P.; McMillan, C.; McBurney, P. W.; Bosch, N.; and D'Mello, S. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*, 390–401. 7

Tipirneni, S.; Zhu, M.; and Reddy, C. K. 2022. StructCoder: Structure-Aware Transformer for Code Generation. *arXiv preprint arXiv:2206.05239*. 1

Tufano, M.; Watson, C.; Bavota, G.; Penta, M. D.; White, M.; and Poshyvanyk, D. 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): 1–29. 4, 10

Wang, A.; Singh, A.; Michael, J.; Hill, F.; Levy, O.; and Bowman, S. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, 353–355. 10

Wang, Y.; Wang, W.; Joty, S.; and Hoi, S. C. 2021. 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*, 8696–8708. 1, 4, 10

Wu, Y.; Schuster, M.; Chen, Z.; Le, Q. V.; Norouzi, M.; Macherey, W.; Krikun, M.; Cao, Y.; Gao, Q.; Macherey, K.; et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*. 3

Yang, P.; Chen, J.; Hsieh, C.-J.; Wang, J.-L.; and Jordan, M. I. 2020. Greedy Attack and Gumbel Attack: Generating Adversarial Examples for Discrete Data. *J. Mach. Learn. Res.*, 21(43): 1–36. 2

Yang, Z.; Shi, J.; He, J.; and Lo, D. 2022. Natural Attack for Pre-Trained Models of Code. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, 1482–1493. New York, NY, USA: Association for Computing Machinery. ISBN 9781450392211. 2

Yefet, N.; Alon, U.; and Yahav, E. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–30. 2

Zhang, H.; Li, Z.; Li, G.; Ma, L.; Liu, Y.; and Jin, Z. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 1169–1176. 2

Zhang, Z.; Zhang, H.; Shen, B.; and Gu, X. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1073–1084. 1, 2, 3, 7

Zhou, Y.; Zhang, X.; Shen, J.; Han, T.; Chen, T.; and Gall, H. 2022. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4): 1–30. 2

A Appendix

Downstream Tasks and Datasets

We evaluate the transferability of CodeAttack across different sequence to sequence downstream tasks and datasets in different programming languages.

- **Code Translation** involves translating one programming language to the other. The publicly available code translation datasets[2] consists of parallel functions between Java and C#. There are a total of 11,800 paired functions, out of which 1000 are used for testing. The average sequence length for Java functions is 38.51 tokens, and the average length for C# functions is 46.16.
- **Code Repair** refines code by automatically fixing bugs. The publicly available code repair dataset (Tufano et al. 2019) consists of buggy Java functions as source and their corresponding fixed functions as target. We use the *small* dataset with 46,680 train, 5,835 validation, and 5,835 test samples ($\leq$ 50 tokens in each function).
- **Code Summarization** involves generating natural language summary for a given code. We use the CodeSearchNet dataset (Husain et al. 2019) which consists of code and their corresponding summaries in natural language. We show the results of our model on Python (252K/14K/15K), Java (165K/5K/11K), and PHP (241K/13K/15K). The numbers in the bracket denote the samples in train/development/test set, respectively.

Victim Models

We pick a representative method from different categories as our victim models to evaluate the attack.

- **CodeT5** (Wang et al. 2021): A unified pre-trained *encoder-decoder* transformer-based PL model that leverages code semantics by using an identifier-aware pre-training objective. This is the state-of-the-art on several sub-tasks in the CodeXGlue benchmark (Lu et al. 2021).
- **CodeBERT** (Feng et al. 2020): A bimodal pre-trained *programming language model* that performs code-code and code-NL tasks.
- **GraphCodeBert** (Guo et al. 2020): Pre-trained *graph programming language model* that *leverages code structure* through data flow graphs.
- **RoBERTa** (Liu et al. 2019): Pre-trained *natural language model* with state-of-art results on GLUE (Wang et al. 2018), RACE (Lai et al. 2017), and SQuAD (Rajpurkar et al. 2016) datasets.

We use the publicly available fine-tuned checkpoints for CodeT5 and fine-tune CodeBERT, GraphCodeBERT, and RoBERTa on the related downstream tasks.

Baseline Models Since CodeAttack operates in the natural channel of code, we compare with two state-of-the-art adversarial NLP baselines for a fair comparison:

- **TextFooler (Jin et al. 2020)**: Uses a combination of synonyms, Part-Of-Speech (POS) checking, and semantic similarity to generate adversarial text.
- **BERT-Attack (Li et al. 2020)**: Uses a pre-trained BERT masked language model to generate adversarial examples satisfying a certain similarity threshold.

Results

Downstream Performance and Attack Quality We measure the CodeBLEU and $\Delta_{CodeBLEU}$ to evaluate the downstream performance for code-code tasks (code repair and code translation). The programming languages used are C#-Java and Java-C# for translation tasks; and Java for code repair tasks (Table 4 and Table 5). We show the results for code-NL task for code summarization in BLEU and Δ_{BLEU}. We show the results for three programming languages: Python, Java, and PHP (Table 4 and Table 6). We measure the quality of the attacks using the metric defined in 4. The results follow a similar pattern as that seen in Sections 4.

Ablation Study: Qualitative Analysis Table 7 shows the adversarial examples generated using the variants described in Section 4.

[2]http://lucene.apache.org/, http://poi.apache.org/, https://github.com/eclipse/jgit/, https://github.com/antlr/

Task	Victim Model	Attack Method	Attack Effectiveness				Attack Quality		
			Before	**After**	Δ_{drop}	**Success%**	**#Queries**	**#Perturb**	**CodeBLEU$_q$**
Translate (C#-Java)	CodeT5	TextFooler	73.99	68.08	5.91	28.29	94.95	2.90	63.19
		BERT-Attack		63.01	10.98	75.83	163.5	5.28	62.52
		CodeAttack		**61.72**	**12.27**	**89.3**	**36.84**	**2.55**	**65.91**
	CodeBERT	TextFooler	71.16	60.45	10.71	49.2	73.91	1.74	66.61
		BERT-Attack		58.80	12.36	70.1	290.1	5.88	52.14
		CodeAttack		**54.14**	**17.03**	**97.7**	**26.43**	**1.68**	**66.89**
	GraphCode-BERT	Textfooler	66.80	46.51	20.29	38.70	83.17	1.82	63.62
		BERT-Attack		**36.54**	**30.26**	94.33	175.8	6.73	52.07
		CodeAttack		38.81	27.99	**98**	**20.60**	**1.64**	**65.39**
Translate (Java-C#)	CodeT5	TextFooler	87.03	79.83	7. 20	32.3	62.91	2.19	**81.28**
		BERT-Attack		68.81	18.22	86.3	89.99	2.79	74.52
		CodeAttack		**66.97**	**20.06**	**94.8**	**19.85**	**2.03**	75.21
	CodeBERT	TextFooler	83.48	73.52	9.96	55.9	38.57	2.14	73.93
		BERT-Attack		67.94	15.5	70.3	159.1	5.76	46.82
		CodeAttack		**66.98**	**16.5**	**91.1**	**24.42**	**1.66**	**76.77**
	GraphCode-BERT	Textfooler	82.40	74.32	8.2	51.2	39.33	2.03	72.45
		BERT-Attack		64.87	17.5	76.6	134.3	5.90	47.47
		CodeAttack		**58.88**	**23.5**	**90.8**	**23.22**	**1.64**	**77.33**
Repair (small)	CodeT5	Textfooler	61.13	57.59	3.53	58.84	90.50	2.36	**69.53**
		BERT-Attack		**52.70**	**8.43**	94.33	262.5	15.1	53.60
		CodeAttack		53.21	7.92	**99.36**	**30.68**	**2.11**	69.03
	CodeBERT	Textfooler	61.33	53.55	7.78	81.61	45.89	2.16	**68.16**
		BERT-Attack		**51.95**	**9.38**	95.31	183.3	15.7	61.95
		CodeAttack		52.02	9.31	**99.39**	**25.98**	**1.64**	68.05
	GraphCode-BERT	Textfooler	62.16	54.23	7.92	78.92	51.07	2.20	**67.89**
		BERT-Attack		53.33	8.83	96.20	174.1	15.7	53.66
		CodeAttack		**51.97**	**10.19**	**99.52**	**24.67**	**1.67**	66.16
Summarize (PHP)	CodeT5	TextFooler	20.06	14.96	5.70	64.6	410.15	**6.38**	**53.91**
		BERT-Attack		11.96	8.70	78.4	1014.1	7.32	51.34
		CodeAttack		**11.06**	**9.59**	**82.8**	**314.87**	10.1	52.67
	CodeBERT	Textfooler	19.76	14.38	5.37	61.10	358.43	2.92	**54.10**
		BERT-Attack		11.30	8.35	56.47	1912.6	15.8	46.24
		CodeAttack		**10.88**	**8.87**	**88.32**	**204.46**	**2.57**	52.95
	RoBERTa	TextFooler	19.06	14.06	4.99	62.60	356.68	2.80	**54.11**
		BERT-Attack		11.34	7.71	60.46	1742.3	17.1	46.95
		CodeAttack		**10.98**	**8.08**	**87.51**	**183.22**	**2.62**	53.03
Summarize (Python)	CodeT5	TextFooler	20.36	12.11	8.25	90.47	400.06	5.26	**77.59**
		BERT-Attack		8.22	12.14	97.81	475.61	6.91	66.27
		CodeAttack		**7.97**	**12.39**	**98.50**	**174.05**	**5.10**	69.17
	CodeBERT	TextFooler	26.17	22.76	3.41	68.50	966.19	3.83	**75.15**
		BERT-Attack		22.88	3.29	84.41	941.94	3.35	56.31
		CodeAttack		**18.69**	**7.48**	**86.63**	**560.68**	**3.23**	59.11
	RoBERTa	Textfooler	17.01	10.72	6.29	63.34	788.25	3.57	70.48
		BERT-Attack		10.66	6.35	74.64	1358.8	4.07	51.74
		CodeAttack		**9.50**	**7.51**	**76.09**	**661.75**	**3.46**	61.22
Summarize (Java)	CodeT5	TextFooler	19.77	14.06	5.71	67.80	291.82	**3.76**	**90.82**
		BERT-Attack		11.94	7.83	77.37	811.97	17.4	45.71
		CodeAttack		**11.21**	**8.56**	**80.80**	**198.11**	7.43	90.04
	CodeBERT	TextFooler	17.65	16.44	1.21	42.4	400.78	4.07	**90.29**
		BERT-Attack		15.49	2.16	46.51	1531.1	10.9	37.61
		CodeAttack		**14.69**	**2.96**	**73.70**	**340.99**	**3.27**	59.37
	RoBERTa	Textfooler	16.47	13.23	3.24	44.9	383.36	4.02	**90.87**
		BERT-Attack		11.89	4.58	42.59	1582.2	9.21	37.86
		CodeAttack		**11.74**	**4.73**	**50.14**	**346.07**	**3.29**	48.48

Table 4: Results on code translation, code repair, and code summarization tasks. The performance is measured in CodeBLEU for Code-Code tasks and in BLEU for Code-NL (summarization) task. The best result is in **boldface**; the next best is underlined.

Original Code	TextFooler	BERT-Attack	CodeAttack

`public string GetFullMessage() {`
`...`
`if (msgB < 0){return string.Empty;}`
`...`
`return RawParseUtils.Decode(enc, raw, msgB, raw.Length);}`
	`citizenship string GetFullMessage() {`
	`...`
	`if (msgB < 0){return string.Empty;}`
	`...`
	`return RawParseUtils.Decode(enc, raw, msgB, raw.Length);}`
		`loop string GetFullMessage() {`
		`...`
		`if (msgB < 0){return string.Empty;}`
		`...`
		`return here q. dir, (x) raw, msgB, raw.Length);}`
			`public string GetFullMessage() {`
			`...`
			`if (msgB = 0){return string.Empty;}`
			`...`
			`return RawParseUtils.Decode(enc, raw, msgB, raw.Length);}`

CodeBLEU$_{\text{before}}$: 77.09	Δ_{drop}: 18.84; CodeBLEU$_q$: 95.11	Δ_{drop}: 15.09; CodeBLEU$_q$: 57.46	Δ_{drop}: 21.04; CodeBLEU$_q$: 88.65

`public override void WriteByte(byte b) {`
`if (outerInstance.upto == outerInstance.blockSize) {... }}`
	`audiences revoked canceling WriteByte(byte b) {`
	`if (outerInstance.upto == outerInstance.blockSize) {.... }}`
		`public override void ; . b) {`
		`if (outerInstance.upto == outerInstance.blockSize) {... }}`
			`public override void WriteByte(bytes b) {`
			`if (outerInstance.upto == outerInstance.blockSize) {... }}`

CodeBLEU$_{\text{before}}$:100	Δ_{drop}:5.74; CodeBLEU$_q$: 63.28	Δ_{drop}:27.26; CodeBLEU$_q$:49.87	Δ_{drop}:20.04; CodeBLEU$_q$: 91.69

Table 5: Qualitative examples of perturbed codes using TextFooler, BERT-Attack, and CodeAttack on Code Translation task.

Original	TextFooler	BERT-Attack	CodeAttack

`protected final void fastPathOrderedEmit(U value, boolean delayError, Disposable disposable) {`
`final Observer<? super V> observer = downstream;`
`final ..... {`
`if (q.isEmpty()) {`
`accept(observer, value);`
`if (leave(-1) == 0) {`
`return;`
`}`
`} else {`
`q.offer(value);`
`}`
`} else {`
`q.offer(value);`
`if (!enter()) {`
`return;`
`}`
`}`
`QueueDrainHelper.drainLoop(q, observer, delayError, disposable, this);`
`}`
	`protected final invalidate fastPathOrderedEmit(U value, boolean delayError, Disposable disposable) {`
	`finalizing Observer < ? super V > observer = downstream;`
	`final .... {`
	`if (q.isEmpty()) {`
	`accept(observer, value);`
	`if (leave(-1) == 0) {`
	`return;`
	`}`
	`}`
	`yet {`
	`q.offer(value);`
	`}`
	`}`
	`annan {`
	`q.offer(value);`
	`than(!enter()) {`
	`return;`
	`}`
	`}`
	`QueueDrainHelper.drainLoop(q, observer, delayError, disposable, this);`
	`}`
		`(; period fastPathOrderedEmit(U value, this)0 c | / , ) ( fore pas Observer<? super V > observer = downstream;`
		`final .... {`
		`if (q.isEmpty()) {`
		`accept() ,point 0while ( leave ... ) ] a ) ] ]returnspublic . next , manager q . offer , value )3̆009 thereforereturn draw q . offer , value ) ... )? enter ) public`
		`}`
		`QueueDrainHelper.drainLoop(q, observer, ) ) 1 0c ) ) , these ?`
		`}`
			`protected final static fastPathOrderedEmit(M value, boolean ZdelayExc, Disposable dis_zyk) {`
			`final Observer <? super V > observer = downstream;`
			`final .... {`
			`if (q.isEmpty()) {`
			`accept(observer, value);`
			`if (leave(-1) == 0) {`
			`continue;`
			`}`
			`} catch {`
			`q.offer(value);`
			`}`
			`} else {`
			`q.offer(value);`
			`if (!exit()) {`
			`return;`
			`}`
			`}`
			`QueueDrainHelper.drainLoop(q, observer, delayInfo, disposable, this);`
			`}`

Makes sure the fast-path emits in order	This method is used to avoid the need for the fast path to emit a value to the downstream.	period 0||||||	dddddsssss

Table 6: Qualitative examples of adversarial codes and the generated summary using TextFooler, BERT-Attack, and CodeAttack on Code Summarization task.

Original Code	**CodeAttack$_{\texttt{VUL}}$**	**CodeAttack$_{\texttt{VUL+OP}}$**	**CodeAttack$_{\texttt{VUL+OP+TOK}}$**

`public void AddMultipleBlanks(`
`      MulBlankRecord mbr) {`
`  for (int j = 0; j < mbr.`
`        NumColumns; j++) {`
`    BlankRecord br = new`
`          BlankRecord();`
`    br.Column = j + mbr.`
`          FirstColumn;`
`    br.Row = mbr.Row;`
`    br.XFIndex = (mbr.GetXFAt(j`
`          ));`
`    InsertCell(br);`
`  }`
`}`
 	
CodeBLEU$_{\text{before}}$: 76.3
	`((void AddMultipleBlanks(`
	`      MulBlankRecord mbr) {`
	`    for (int j ? 0; j < mbr.`
	`          NumColumns; j++) {`
	`      BlankRecord br = new`
	`            BlankRecord();`
	`      br.Column = j + mbr.`
	`            FirstColumn;`
	`      br.Row = mbr.Row;`
	`      br.XFIndex = (mbr.`
	`            GetXFAt(j));`
	`      InsertCell(br);`
	`    }`
	`  }`
	
	Δ_{drop}: 7.21; CodeBLEU$_q$: 43.85
		`public void AddMultipleBlanks(`
		`      MulBlankRecord mbr) {`
		`  for (int j > 0; j < mbr.`
		`        NumColumns; j++) {`
		`    BlankRecord br = -new`
		`          BlankRecord();`
		`    br.Column = j + mbr.`
		`          FirstColumn;`
		`    br.Row = mbr.Row;`
		`    br.XFIndex > (mbr.GetXFAt(j`
		`          ));`
		`    InsertCell(br);`
		`  }`
		`}`
		
		Δ_{drop}: 5.85; CodeBLEU$_q$: 69.61
			`static void AddMultipleBlanks(`
			`      MulBlankRecord mbr) {`
			`  for (int j > 0; jj < mbr.`
			`        NumColumns; j++) {`
			`    BlankRecord br = new`
			`          BlankRecord();`
			`    br.Column = j + mbr.`
			`          FirstColumn;`
			`    br.Row = mbr.Row;`
			`    br.XFIndex = (mbr.GetXFAt(j`
			`          ));`
			`    InsertCell(br);`
			`  }`
			`}`
			
			Δ_{drop}: 12.96; CodeBLEU$_q$: 59.29

`public string GetFullMessage()`
`      {`
`  byte[] raw = buffer;`
`  int msgB = RawParseUtils.`
`        TagMessage(raw, 0);`
`  if (msgB < 0) {`
`    return string.Empty;`
`  }`
`  Encoding enc = RawParseUtils.`
`        ParseEncoding(raw);`
`  return RawParseUtils.Decode(`
`        enc, raw, msgB, raw.`
`        Length);`
`}`
 	
CodeBLEU$_{\text{before}}$:77.09
	`Ŏ120public string`
	`      GetFullMessage() {`
	`  byte[] raw = buffer;`
	`  int msgB = RawParseUtils.`
	`        TagMessage(raw, 0);`
	`  if (msgB < 0) {`
	`    return string.Empty;`
	`  }`
	`  Encoding enc = RawParseUtils.`
	`        ParseEncoding(raw);`
	`  return RawParseUtils.Decode(`
	`        enc, RAW.., msgB, raw.`
	`        Length);`
	`}`
	
	Δ_{drop}: 10.42; CodeBLEU$_q$: 64.19
		`public string GetFullMessage()`
		`      {`
		`  byte[] raw = buffer;`
		`  int msgB = RawParseUtils.`
		`        TagMessage(raw, 0);`
		`  if (msgB = 0) {`
		`    return string.Empty;`
		`  }`
		`  Encoding enc = RawParseUtils.`
		`        ParseEncoding(raw);`
		`  return RawParseUtils.Decode(`
		`        enc, raw, msgB, raw.`
		`        Length);`
		`}`
		
		Δ_{drop}: 21.93; CodeBLEU$_q$: 87.25
			`static string GetFullMessage()`
			`      {`
			`  byte[] raw = buffer;`
			`  int msgB = RawParseUtils.`
			`        TagMessage(raw, 0);`
			`  if (msgB < 0 {`
			`    return string.Empty;`
			`  }`
			`  Encoding enc = RawParseUtils.`
			`        ParseEncoding(raw);`
			`  return RawParseUtils.Decode(`
			`        enc, raw, MsgB,raw.`
			`        Length);`
			` }`
			
			Δ_{drop}: 22.8; CodeBLEU$_q$: 71.30

Table 7: Qualitative examples for the ablation study on CodeAttack: Attack vulnerable tokens (VUL); with operator level constraints (VUL+OP), and with token level (VUL+OP+TOK) contraints on code translation task.