Adversarial Attacks on Code Models with Discriminative Graph Patterns

Thanh-Dat Nguyen
University of Melbourne
Melbourne, Australia
thanhdatn@student.unimelb.edu.au

Yang Zhou
Singapore Management University
Singapore, Singapore
zyang@smu.edu.sg

Xuan-Bach D. Le
University of Melbourne
Melbourne, Australia
back.le@unimelb.edu.au

Patanamon (Pick) Thongtanunam
University of Melbourne
Melbourne, Australia
patanamon.t@unimelb.edu.au

David Lo
Singapore Management University
Singapore, Singapore
davidlo@smu.edu.sg

ABSTRACT

Pre-trained language models of code are now widely used in various software engineering tasks such as code generation, code completion, vulnerability detection, etc. This, in turn, poses security and reliability risks to these models. One of the important threats is *adversarial attacks*, which can lead to erroneous predictions and largely affect model performance on downstream tasks, necessitating a thorough study of adversarial robustness on code models. Current adversarial attacks on code models usually adopt fixed sets of program transformations, such as variable renaming and dead code insertion. Additionally, expert efforts are required to handcraft these transformations and are limited in terms of creating more complex semantic-preserving transformations.

To address the aforementioned challenges, we propose a novel adversarial attack framework, GRAPHCODEATTACK, to better evaluate the robustness of code models. Given a target code model, GRAPHCODEATTACK automatically mines important code patterns, which can influence the model's decisions, to perturb the structure of input code to the model. To do so, GRAPHCODEATTACK uses a set of input source codes to probe the model's outputs. From these source codes and outputs, GRAPHCODEATTACK identifies the *discriminative* ASTs patterns that can influence the model decisions. GRAPHCODEATTACK then selects appropriate AST patterns, concretizes the selected patterns as attacks, and inserts them as dead code into the model's input program. To effectively synthesize attacks from AST patterns, GRAPHCODEATTACK uses a separate pre-trained code model to fill in the ASTs with concrete code snippets. We evaluate the robustness of two popular code models (e.g., CodeBERT and GraphCodeBERT) against our proposed approach on three tasks: Authorship Attribution, Vulnerability Prediction, and Clone Detection. The experimental results suggest that our proposed approach significantly outperforms state-of-the-art approaches in attacking code models such as CARROT and ALERT. Based on the average attack success rate (ASR), GRAPHCODEATTACK achieved 30% improvement over CARROT and 33% improvement over ALERT respectively. Notably, in terms of ASR on GraphCode-BERT model and on Authorship Attribution, GRAPHCODEATTACK achieved an ASR of 0.841, significantly outperforming CARROT and ALERT (with ASR of 0.598 and 0.615 respectively).

KEYWORDS

Pre-trained Language Model of Code, Adversarial Attack, Discriminative Subgraph Mining

1 INTRODUCTION

Code models [11, 23, 39], especially those built on advanced deep learning architectures, have become increasingly popular recently due to their ability to effectively comprehend programming languages by learning from large-scale code data [13, 16, 49]. These models have been employed and demonstrated strong performance in various applications, including code completion [27], vulnerability detection [56], authorship attribution [4], and code clone detection [29]. Despite their success, recent studies have shown that code models are not robust to *adversarial perturbations* [50, 53] – i.e., semantic-preserving transformations (e.g., renaming the variables or adding some dead code) of the input, that make a code model change predictions from correct to wrong.

The vulnerability of code models to adversarial perturbations can have serious implications for the security and reliability of downstream tasks that employ these models. Consider a scenario where a code model is integrated into an open-source library's contribution review process to detect and prevent the inclusion of vulnerable or malicious code. In this situation, ill-intentioned actors could craft adversarial perturbations to exploit the weaknesses of the code model, thereby causing it to falsely accept their malicious contributions [24]. As a consequence, the library could unknowingly incorporate security vulnerabilities or harmful code, leading to significant risks for the library's users and potentially damaging the reputation of the project maintainers. This threat model highlights the importance of assessing and enhancing the robustness of code models against adversarial attacks.

In this paper, we evaluate the code model robustness in a black-box setting: the attacker only has access to the model's output and cannot access the internal information (e.g., parameter and gradient information) of the victim model, nor the ground truth label. The black-box assumption is realistic as such code models are usually deployed remotely and can be accessed by APIs. Many recent works [19, 50, 52, 53] also adopt the same assumption. ALERT [50] and CARROT [53] are state-of-the-art techniques for adversarial attacks on code models. They focus on using a fixed set of hand-crafted patterns to transform the inputs of code models. ALERT [50] uses variable renaming and CARROT [53] uses additional transformations, e.g., adding dead-code with manually-designed patterns such as while(false), if(false), etc. Attacking code models using hand-crated transformations, however, presents certain limitations. Particularly, the hand-crafted patterns may not stay abreast of fastly growing datasets to adequately represent the diverse range of real-world code structures and may have limitations in modeling complex semantic-preserving transformations. Hence, there is a need for an automated and systematic process of identifying potential adversarial attacks, which will enable the developers thoroughly test and assure the reliability of the code model before releasing it to the users.

We introduce GRAPHCODEATTACK, a novel approach to attack models of code by using *automatically mined* code patterns that can highly influence a target model's decisions. Doing so allows GRAPHCODEATTACK to flexibly adapt to different code models with varying training data, as opposed to the use of handcrafted transformations by current state-of-the-art approaches. Given a target model of code and a set of probing data (i.e., a set of data used to test the model's output), GRAPHCODEATTACK works in three phases: mining *highly influential* patterns, synthesizing attacks from patterns, and selecting appropriate attacks.

GRAPHCODEATTACK first automatically identifies *discriminative* AST patterns from the probing data (i.e., programs' source code) that highly influence the target model's prediction. To achieve this, we employ a discriminative subgraph mining technique, namely the gspan-CORK algorithm [46]. This allows us to find frequent subgraphs or patterns in the data that are discriminative between different classes or groups of data. Second, GRAPHCODEATTACK synthesizes concrete attacks based on the patterns mined in the previous step. Note that the mined AST patterns primarily contain structural information, such as node types and edge types, without any actual concrete content (e.g., specific identifiers or particular binary operations among expressions like +, −, etc.). To fill this gap, we leverage a language model, which is different from the model under attack, to synthesize concrete code from abstract patterns. Large language models, such as CodeBERT [13] and CodeT5 [8], have demonstrated capabilities in completing code spans that are contextually coherent. We thus leverage these models for the attack synthesis step. GRAPHCODEATTACK then searches through the synthesized concrete attacks to find appropriate attacks to be inserted into a given program as input to the target model.

We evaluate the robustness of two popular code models (e.g., CodeBERT and GraphCodeBERT) against our proposed approach on three tasks: Authorship Attribution, Vulnerability Prediction, and Clone Detection. Experiments suggest that our proposed approach significantly outperforms state-of-the-art approaches in

code model attacks such as CARROT and ALERT. Based on the average attack success rate (ASR), GRAPHCODEATTACK achieved 30% improvement over CARROT and 33% improvement over ALERT respectively. Notably, on the GraphCodeBERT model and on Authorship Attribution, GRAPHCODEATTACK achieved a score of 0.84 in terms of average ASR.

In summary, we present GRAPHCODEATTACK, a novel approach for attacking models of code by synthesizing adversarial examples from attack patterns that are automatically mined from a set of probing data and the corresponding model's output. Our contributions can be summarized as follows:

- Our novel approach GRAPHCODEATTACK automatically mines attack patterns from a model and the set of probing data, rendering the derived attacks flexibly adaptable to specific target models and domains.
- We introduce a novel method that leverages pre-trained language models to automatically generate effective concrete attacks from discovered abstract AST patterns. In comparison with CARROT's random identifier renaming and ALERT's code-model-based identifier renaming, GRAPHCODEATTACK surpasses their performance in 5 out of the 6 task-and-model combinations evaluated.
- We demonstrate the effectiveness of our approach through extensive experiments, showing that GRAPHCODEATTACK can successfully synthesize adversarial examples that challenge the robustness and reliability of code models. Particularly, GRAPHCODEATTACK achieved 30% improvement over CARROT and 33% improvement over ALERT on average ASR measurement. Notably, on the GraphCodeBERT model, GRAPHCODEATTACK achieved 0.84 and 0.799 in terms of ASR on Authorship Attribution and Vulnerability Prediction respectively.

The rest of the paper is organized as follows: Section 2 provides background on code models, adversarial attacks, and abstract syntax trees. Section 3 details the proposed GRAPHCODEATTACK methodology, including the process of mining AST patterns, and the usage of pre-trained language models for pattern insertion. Section 7 presents the related works on the problem of adversarial attack on the model of code. Sections 4 and 5 describe our experiment settings and the results respectively.

2 BACKGROUND

In this section, we provide essential background information on code models and the use of Abstract Syntax Trees (ASTs) for graph mining techniques that form the basis of our proposed system architecture for attacking code models.

2.1 Code Models

Code models [13, 16] are machine learning models designed to analyze, understand, and generate source code. They play a crucial role in various software engineering tasks, such as code completion, code summarization, bug detection, and vulnerability identification. Recent advances in deep learning have led to the development of more sophisticated code models, such as Transformer-based models, that can capture complex patterns and structures in source

code. These transformer models can be pre-trained using unla-belled code datasets to capture the semantic relations in the source code [13, 16]. After pre-training, these models can be fine-tuned to achieve state-of-the-art performance on downstream tasks such as code completion, vulnerability prediction, authorship attribution, etc. [10, 29]. However, these models are also susceptible to adver-sarial attacks, where carefully crafted perturbations in the input source code can cause them to produce incorrect predictions or outputs [50, 53].

2.2 Graph Mining via Abstract Syntax Trees

Abstract Syntax Tree. Abstract Syntax Trees (ASTs) represent the syntactic structure of source code, with nodes corresponding to language constructs and edges indicating the relationships between nodes. In our context, we describe the ASTs as graphs. In detail, an AST is denoted as a graph $G = (V, E)$, where the vertex set V consists of nodes corresponding to language constructs, and the edge set E includes directed edges indicating their parent-child rela-tionships. Each node $n \in V$ is associated with a label l_n representing the language construct it denotes, such as variables, expressions, or control structures. Similarly, each edge $e \in E$ can also have a corresponding label l_e specifying the relationship between the con-nected nodes, such as data or control dependencies. For example, in an AST representing a simple if-else statement, the nodes might represent the if keyword, the condition, and the branches, while the edges indicate the parent-child relationships among these nodes. The structure of the AST captures the hierarchical organization and the syntactic dependencies in the source code, which are essential for mining discriminative patterns.

Discriminative Patterns Mining. Discriminative subgraph min-ing [46] is a branch of graph mining that focuses on discovering subgraphs or patterns that exhibit significant differences between classes in the dataset. The goal is to identify the most distinguishing substructures for each class, which can then be used for tasks such as classification, clustering, and anomaly detection. Since these subgraphs are discriminative, their presence might be more likely to change the prediction of the target model. Thus, GRAPHCODEAT-TACK perturb the input source code by inserting these patterns.

GRAPHCODEATTACK works by finding the most discriminative subgraphs from a dataset and uses these subgraphs to structurally perturb the input code. Our GRAPHCODEATTACK analyzes ASTs since ASTs provide a more structured and semantically rich repre-sentation of source code than raw text, facilitating pattern mining and analysis. Also, ASTs are a generic representation, allowing our approach to be applicable across different programming languages.

3 METHODOLOGY

In this section, we present the architecture of GRAPHCODEATTACK. GRAPHCODEATTACK aims at deriving adversarial attacks of models of code by automatically mining attack patterns from a set of prob-ing data and the corresponding model's output. The attack patterns are in the form of abstract syntax trees (ASTs) that can be used to perturb the structure of a model's input code which influence the model's decisions.

GRAPHCODEATTACK has to overcome three primary challenges: (1) How to automatically obtain the set of effective abstract patterns

that are flexibly adaptable to each model using a set of probing data (2) How to effectively derive concrete attacks from an abstract pattern, and (3) Which patterns to select and where to insert it into the input code.

To address these challenges, GRAPHCODEATTACK operates in three main phases. First, GRAPHCODEATTACK formulates the prob-lem of identifying effective AST patterns as *discriminative subgraph mining* (the mining phase 3.1). Taking a set of input code $\mathcal{D}$ and the corresponding model's predictions as input, GRAPHCODEATTACK identifies a set of discriminative abstract syntax tree patterns $\mathcal{P}_A$ that are highly correlated with the model's predictions. This means that the presence or absence of these patterns is closely linked to specific predictions made by the models based on the preprocessing phase. Based upon this link, GRAPHCODEATTACK tries to alter the model prediction by inserting these patterns into the input source code. Note that these patterns are inserted to a program in a way that the semantics of the underlying program remain intact.

Recall that the mined AST patterns are abstract, only containing information such as node type, edge types, etc, without concrete content (e.g., specific identifier, specific binary, unary operations like +, -, >, <, etc.). To synthesize concrete attacks from the abstract AST patterns, GRAPHCODEATTACK convert each pattern into a textual form, in which parts that need to be filled in are indicated by a special token <MASK>. This textual representation help facilitate the insertion of the pattern in the attack phase.

Finally, the attack phase focuses on determining the valid pertur-bation of the input source code that makes the target model change prediction. To search for this perturbation, GRAPHCODEATTACK for-mulates the problem as a search problem of positions in the source code and the corresponding modifications to perform at each corre-sponding position. We sample the positions based on a calculated important score, which specifies the effectiveness of having the statement and not having the statement on the target model's pre-diction. Followed by that, we estimate the most impactful pattern to insert based on a meta-model over the model's output. Having determined the position as well as the corresponding patterns, we insert these patterns into the input source code.

To perform the attack, we implement a greedy strategy. At each greedy step, we choose the position/pattern combination that can most reduce the confidence of the target model on its output. This assumption has also been adopted by earlier works [50, 53]. We keep track of the number of model queries and stop this process once the maximum number of queries is reached.

We explain in detail the three main phases of GRAPHCODEAT-TACK in the below subsections.

3.1 Mining Attack Patterns

In this section, we detail the process of mining attack patterns in GRAPHCODEATTACK. The primary objective is to identify a set of discriminative subgraphs $\mathcal{P}_A$ that can effectively influence the target model's prediction.

3.1.1 Discriminative Subgraph Mining. Given a set of input code samples $S = s_1, s_2, \ldots, s_n$ and their corresponding model predictions $Y = y_1, y_2, \ldots, y_n$, we formulate the task of finding effective pat-terns to attack the model as a discriminative subgraphs mining prob-lem. The goal is to discover a set of subgraphs $\mathcal{P}_A = P_1, P_2, \ldots, P_k$
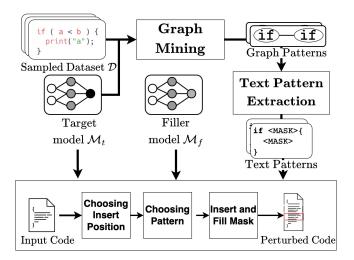
Figure 1. Overview of GraphCodeAttack's method. $\mathcal{M}_t$ is the target victim model, $\mathcal{M}_f$ is the language model used to fill in the `<MASK>`

that are significantly discriminative. These subgraphs, which discriminate between different classes or groups of data, empower us to effectively sway the model's prediction.

To achieve this, we first construct each AST representation T_i for each code sample $s_i \in S$. After obtaining the resulting ASTs set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$, we apply the gSpan-CORK algorithm to find the set of subgraphs $\mathcal{P}_A$ that exhibit a high discriminative power. gSpan-CORK aims to greedily find the the set of subgraphs $\mathcal{P}_A = \{P_{A_1}, P_{A_2}, \ldots, P_{A_k}\}$ that maximizes the quality criterion q. For two classes 0 and 1 with the corresponding set of ASTs $\mathcal{T}_0$ and $\mathcal{T}_1$:

$$q(\mathcal{P}_A) = -\sum_{P \in \mathcal{P}_A} \left(|\mathcal{T}_{0,\bar{P}}| \cdot |\mathcal{T}_{1,\bar{P}}| + |\mathcal{T}_{0,P}| \cdot |\mathcal{T}_{1,P}|\right) \tag{1}$$

where $\mathcal{T}_{0,\bar{P}}$, $\mathcal{T}_{1,\bar{P}}$ are the sets of ASTs belonging to class 0 and class 1 that do not contain subgraph P. $\mathcal{T}_{0,P}$, $\mathcal{T}_{1,P}$ is the set of ASTs belonging to class 0 and class 1 that contains P respectively. The intuition behind this function is to maximize the difference in subgraph patterns between the two classes.

The first term counts the cases where subgraph P is absent in both class 0 and class 1, while the second term counts the cases where subgraph P is present in both classes. By minimizing the sum of these cross-interactions, we aim to find the set of subgraphs $\mathcal{P}_A$ that best differentiates the two classes. These subgraphs provide the basis for the subsequent attack phase, where they are used to perturb the source code and alter the model's output. We note that there can be cases where the model can perform multi-class classifications that predict the output to be one of C classes. In this case, we simply construct C one-versus-all graph datasets.

3.2 Synthesizing Concrete Attacks from AST Patterns

Recall that each pattern P in the discriminative AST pattern set $\mathcal{P}_A$ is a subgraph. Utilizing these patterns to guide the perturbation of code presents some challenges, as the process is not straightforward.

Particularly, as the mined graphs are abstract, we need to synthesize concrete code snippets from the patterns in a way that the snippets are contextually coherent with the underlying program.

Figure 3 demonstrates a motivating example. We have an attack pattern consisting of a binary operation with the left side component pointing to a string of unknown content and the right side component pointing to an unknown node. There are several problems with inserting this pattern into the code: (1) Filling in the content of the right node is problematic as we do not know the actual type, name, or value of the node. (2) We do not know what exactly the operation of **Binary Op** node and (3) Furthermore, we also have to consider what variable or expression is based on each context to be put in the right side. GraphCodeAttack tackles these challenges using a pre-trained model of code. As an example, consider the abstract syntax subtree depicted in Figure 3.

To determine which operations can be put in **Binary Op** node, we identify the corresponding instance of the pattern in the actual source code and narrow down the set of values that can be put in. Having identified the instances of the pattern in the dataset, we also know how to identify the components that can be changed. For incomplete components (e.g., string without string content, a right node of the binary component), we identify the textual span of the corresponding component and replace its textual content with the special `<MASK>` token.

The result is the textual representation of the mined subgraph with unknown components replaced by the `<MASK>` token. As we will see later, this representation facilitates the insertion of the pattern into the source code using the pre-trained language models.

3.3 Attacking with mined patterns

Having obtained the pattern and the corresponding textual representation, we proceed to perform the adversarial attack on the target model. Taking a source code as an input, GraphCodeAttack repeatedly chooses the most important statements along with a pattern that likely impacts the model prediction and inserts the pattern next to the statement until it reaches the token threshold limit. We give an illustration in Figure 2.

3.3.1 Statement-level important score estimation. To choose the most important statement, we quantify the impact of a statement by computing the difference in the target model's probability estimates with and without the inclusion of the statement. Let $P_{\mathcal{M}_t, c_t}(s)$ denote the target model $\mathcal{M}_t$'s output probability for a given input code sample s and for the target class c_t. Let a be a statement in s, the impact of a on s is:

$$\Delta P_{\mathcal{M}_t, c_t}(s, a) = P_{\mathcal{M}_\sqcup, c_t}(s) - P_{\mathcal{M}_t, c_t}(s \setminus a) \tag{2}$$

Where $s \setminus a$ represents the input code sample s without the statement a included. Intuitively, the larger $\Delta P_{\mathcal{M}_t, c_t}(s, a)$ is the more positive impact of the statement towards the prediction of the model $\mathcal{M}_t$. We employ a greedy strategy and choose the most important statement as the attack location.

3.3.2 Choosing pattern with meta-model. Since the number of patterns can be large, the next question is how to choose an effective attack pattern that would likely lead to a different prediction of the model. For this, we train a decision tree as a meta-model
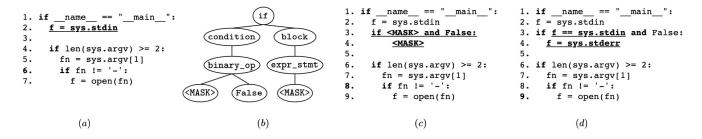
Figure 2. Attacking with pattern: Given the original source code (a), GRAPHCODEATTACK identify the important statement on line 2: `f = sys.stdin`. GRAPHCODEATTACK then chooses the pattern (b) consisting of an if statement with unknown condition and body. GRAPHCODEATTACK inserts this text pattern in the code, resulting in the masked code (c). Finally, GRAPHCODEATTACK uses the filler language model $\mathcal{M}_f$ to fill in the mask in (c), resulting in the perturbed code (d) that changes model prediciton
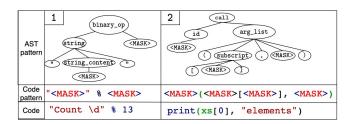

Figure 3. Example of corresponding AST pattern and textual pattern

$\mathcal{M}_{meta} : s \mapsto y$. This meta-model is trained to predict the target model prediction y, given the presence of each pattern in the pattern set $\mathcal{P}_A$. In detail, as input to the meta-model, we use a bag-of-pattern encoding.

Obtaining the meta-model For each sample source code s_i, we construct the feature $\mathbf{f}_i \in \{0, 1\}^{|\mathcal{P}_A|}$. Where:

$$f_{i,j} = \begin{cases} 1 & \text{if } s \text{ contains the pattern } P_j \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Given the features f_i and the corresponding prediction y_i for each source code s_i, we can train a decision tree $\mathcal{M}_{meta}$. Each path π in this decision tree corresponds to a predicted class c_π and the number of support SP_π (i.e., the number of samples in $\mathcal{D}$ that contains the patterns indicated in the path and received model's prediction to be c_π).

Choosing pattern Given the information on c_π and SP_π of each path π, we can now determine which pattern to be inserted into the target source code. For each new input s_{new} with AST T_{new}, we identify its corresponding features f_{new}. Furthermore, for each missing pattern P_{miss} in the set of missing patterns $\mathcal{P}_{miss} = \{P_i \in \mathcal{P}_A | P_i \nsubseteq T_{new}\}$, we calculate the approximated probability that adding this missing pattern leads to a different prediction using the meta-model $\mathcal{M}_{meta}$.

$$P(\mathcal{M}_{meta}(T_{new} \cup P_i) \neq y_i) = \frac{\sum_\pi SP_\pi \times (\mathbb{1}_{c_\pi \neq y_i})}{n} \quad (4)$$

Where $\mathbb{1}_{c_\pi \neq y_i} = \begin{cases} 1 & \text{if } c_\pi \neq y_i \\ 0 & \text{otherwise} \end{cases}$ is the indicator function.

We sample the pattern with the probability of changing model

prediction:

$$P_{chosen} = \underset{P_{miss}}{\arg\max} P(\mathcal{M}_{meta}(T_{new} \cup P_{miss}) \neq y_i) \quad (5)$$

3.3.3 Pattern insertion. Having determined the location and pattern to insert, the final question is how to insert the pattern and fill in the <MASK> token such that the final code is syntactically valid and semantically preserving. To generate syntactically valid code, we leverage a different pre-trained language model $\mathcal{M}_f$ (namely, we use the language-specific CodeBERT provided from CodeBERTScore[55]) to fill in the <MASK>. Note that this model is separated from the target model $\mathcal{M}_t$. In order to make sure the code does not change the semantics of the current code, we follow CARROT's S-modifier [53] (which either inserts dead statement or wraps the code inside a redundant branching) and modify the pattern:

- If the pattern is conditional (e.g., `while` loop, `for` loop, `if` condition, etc, we modify the original condition of the pattern from <MASK> to `false && (<MASK>)`).
- Else, we put the pattern inside a dead code block.

Finally, we insert the modified text pattern into the chosen location and use a pre-trained language model to fill in the mask and obtain the perturbed code. We note that the pre-trained language model might not always generate syntactically valid code. Therefore, we employ tree-sitter parser[1] to re-parse the generated code and to check if there exist errors in the filled source code. If the tree-sitter detects an erroneous node (i.e., the node has the label "ERROR"), we retry generating the node up to 5 times then discard the candidate, else, we query the target model to obtain the new evaluation. The patterns are inserted until either the number of maximum target model queries is met, or the target model changes its prediction.

4 EXPERIMENT SETTINGS

The experiment results reported here were obtained on an Intel i5-9600K machine with 64 GB of RAM and equipped with one Nvidia GTX 1080Ti running Linux.

---

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

4.1 Dataset

In our experiment, we follow the settings of the previous study [50] and select three downstream tasks from the CodeXGLUE benchmarks [29]: Vulnerability Prediction, Clone Detection, and Authorship Attribution. Below we introduce the details of each task and its corresponding dataset.

Vulnerability Prediction. The objective of this task is to produce a label indicating whether a specified code snippet contains any vulnerabilities. Zhou et al. [56] label source code in two popular open-sourced C projects: FFmpeg[2] and Qemu[3] to build a dataset consisting of 27,318 functions. Each function is labeled as either containing vulnerabilities or clean. This dataset is widely used to investigate the effectiveness of various code models in understanding code to predict vulnerability. We follow the settings in CodeXGLUE to divide the dataset into training, development, and test sets.

Clone Detection Clone detection is also modeled as a classification problem: given a pair of two code snippets, a code model should predict whether they are clones (i.e., whether they implement the same function). We choose BigCloneBench [45] as the dataset, which is used in the previous study [50] and is a widely-acknowledged benchmark for clone detection. BigCloneBench comprises around 10 million pairs of Java code snippets; over 6 million of them are clones and the remaining 260,000 are not clones. We create a subset of the dataset that has balanced labels (i.e., the ratio of clone pairs and non-clone pairs is 1:1). Following the previous study, we randomly select 90,102 examples for training and 4,000 for validating and testing the code models.

Authorship Attribution The task of authorship attribution involves determining the author of a given code snippet. We choose the Google Code Jam (GCJ) dataset, which is created using the submission from the Google Code Jam challenge, a yearly global coding competition hosted by Google. GCJ dataset is collected and made open-source by Alsulami et al. [4], which consists of 700 Python files that are written by 70 authors. The dataset is balanced, i.e., each author (i.e., class) having 10 code snippets. This dataset contains mainly Python files but also some C++ code. We remove C++ source code to obtain 660 Python files. We follow an 80:20 split: 20% of files are used for testing, and 80% of files are for training. In accordance with previous studies [50], we do not use a validation dataset due to the small dataset size [50].

4.2 Target Model, Filler model and Probing Data

Following the existing works of ALERT [50], we use CodeBERT [13] and GraphCodeBERT [16] as our target models. We follow ALERT [50] in the hyperparameter settings for these models and retrieve the corresponding models from the official GitHub site of ALERT. For the filler model $\mathcal{M}_f$ which is responsible to fill in the mask, we use language-specific CodeBERTs provided by CodeBERTScore [55] which has been pre-trained specifically for each language Python, Java and C respectively. For the probing dataset $\mathcal{D}$, we use each task's training source code without the original label.

Table 1. Statistics of tasks and datasets investigated in the paper, as well as the victim models' performance on these datasets. CB and GCB represent CodeBert and GraphCode-BERT, respectively.

Tasks	Train/Dev/Test	Class	Lang	Model	Acc.
Vulnerability Prediction	21,854/2,732/2,732	2	C	CB	63.76%
				GCB	63.65%
Clone Detection	90,102/4,000/4,000	2	Java	CB	96.97%
				GCB	97.36%
Authorship Attribution	528/-/132	66	Python	CB	90.35%
				GCN	89.48%

4.3 Baselines

In this study, we compare GraphCodeAttack with two state-of-the-art techniques attacking deep code models: CARROT [53] and ALERT [50]. Since CARROT [53] only supports Python, we extend it to attack Python and Java code for sufficient comparison. Furthermore, CARROT has 4 variants: renaming variables with the model's gradient and by random and inserting dead code guided by the target model's gradient and by random. Since we follow the black-box settings in our threat model, we use CARROT I-RW (i.e., random identifier renaming) which is the top-performing candidate towards transformer-based models [53].

5 RESEARCH QUESTIONS.

To investigate GraphCodeAttack against the baselines, we pose three main research questions: (1) The effectiveness and the stealthiness of GraphCodeAttack against CARROT and ALERT, (2) Which patterns are the most effective on each task and model, and (3) How adversarial retraining using GraphCodeAttack compares with ALERT and CARROT on defending against adversarial attacks. We explain each research question and results below.

5.1 RQ1. How effective and stealthy is GraphCodeAttack against the state-of-the-art baselines?

Effectiveness. We use the ALERT's published model as the target model for attacking and compare the Attack Success Rate (ASR) of GraphCodeAttack versus CARROT [53] and ALERT [50] on the 3 tasks: Authorship Attribution, Vulnerability Prediction, and Clone Detection of CodeXGLUE [29]. We use the same number of steps and settings to be 2000 following ALERT [50]. Since GraphCodeAttack's approach needs to execute two code models (the target and the filler) at the same time, it can be slower than the baselines. Thus, in order to give a realistic time efficiency constraint for GraphCodeAttack, we also put a timeout of 100 seconds for each attack. We these results below.

Table 2 presents the ASR of GraphCodeAttack compared to CARROT and ALERT on the CodeXGLUE benchmarks. On average across the three tasks, in terms of attack success rate (ASR), GraphCodeAttack outperforms CARROT [53] by 30.6% and ALERT by 33.1% respectively.

Table 2. Attack Success Rate (ASR) comparison for GRAPHCODEATTACK, CARROT, and ALERT on CodeXGLUE benchmarks for 3 tasks and 2 models (CodeBERT and GraphCodeBERT). Higher values indicate better performance.

Method	Authorship Attribution		Vulnerability Prediction		Clone Detection	
	CodeBERT	GraphCodeBERT	CodeBERT	GraphCodeBERT	CodeBERT	GraphCodeBERT
GRAPHCODEATTACK	**0.612**	**0.8407**	**0.774**	**0.799**	**0.401**	0.053
CARROT [53]	0.485	0.598	0.620	0.746	0.108	**0.102**
ALERT [50]	0.337	0.615	0.536	0.769	0.273	0.080

In detail, for the target model CodeBERT, GRAPHCODEATTACK outperforms CARROT by 26%, 24% and ALERT by 81.5%, 43.6% on Authorship Attribution and Vulnerability Prediction respectively. For GraphCodeBERT, the corresponding improvements are 40.5% and 7% for CARROT and 36.65% and 3.8% for ALERT. For Clone Detection and on CodeBERT, GRAPHCODEATTACK outperformed ALERT by 46.9% and CARROT by 270% respectively. On GraphCodeBERT, GRAPHCODEATTACK performs comparably with CARROT and ALERT. This better performance can be attributed to GRAPHCODEATTACK's capability in leveraging specific attack patterns towards the target model. Pre-trained language models of code rely on both syntactic patterns and textual tokens [22]. The key difference between GRAPHCODEATTACK, ALERT, and CARROT is that GRAPHCODEATTACK adds varied code fragments, while ALERT and CARROT I-RW only change existing variable names. Since variables are only part of the model input, GRAPHCODEATTACK's adding new varying code fragments expands the attack space and results in more comprehensive model perturbation.

The performance of all attack methods changes when switching from CodeBERT to GraphCodeBERT: On GraphCodeBERT [16], GRAPHCODEATTACK marginally outperforms ALERT [50] and CAR-ROT [53], which can be attributed to two factors: (1) GraphCode-BERT's emphasis on variable names, and (2) the perturbations caused by the three tools. Recall that GraphCodeBERT augments CodeBERT with explicit variable names and attention masks, making CARROT and ALERT's renaming more effective. At the same time, since GraphCodeBERT's dataflow does not filter out dead branches, GRAPHCODEATTACK attacks leveraging surrounding variables can still alter the dataflow graph and slightly outperform the baselines.

While the ASR on CodeBERT clone detection by GRAPHCODEATTACK is nearly 42% better than ALERT and four times better than CARROT (i.e., success rates of 0.4, 0.27, 0.1 respectively). The success rates on GraphCodeBERT are low. This suggests that there is still room for improvements on the code clone detection task.
Stealthiness. "Stealthiness" measure how hard it is for the developer to notice the attack. Intuitively, the closer the resulting attacked source code is to the original source code, the harder it is to notice, hence, the attack is stealthier. As a proxy to measure stealthiness automatically, we use *Code Change Rate*. The lower the code change rate is, the stealthier the attacks. Assume that each source codes s is tokenized into n_t number of tokens, and the attack required n_i tokens to be modified. We calculate the average and standard deviation μ_{TC} and σ_{TC} of the number of inserted tokens as well as the change rate of n_i/n_t.

The change rate of GRAPHCODEATTACK is shown in Table 3. On all 3 tasks, GRAPHCODEATTACK needs to insert approximately 100 tokens. For CodeBERT model, GRAPHCODEATTACK inserts on average 90.27 tokens with a standard deviation of 58.38 for Authorship Attribution, 56.33 (38.96) for Vulnerability Prediction, and 125.29 (72.98) for Clone Detection. The corresponding change rates are 0.136, 0.570, and 0.1455. On GraphCodeBERT, GRAPHCODEATTACK has average change rates of 112.833, 62.06, and 124 inserted tokens with standard deviations of 64.4, 47.165, and 70.2 for the three tasks respectively. The corresponding change rates are 0.1359, 0.463, and 0.144. GRAPHCODEATTACK's number of average inserted tokens is smaller in comparison with the number of changed tokens from ALERT and CARROT, as well as having lower variation in the number of inserted tokens across all tasks. Moreover, GRAPHCODEATTACK's change rate is comparable to ALERT and CARROT on Authorship attribution and Clone Detection but is higher in Vulnerability Detection. This is due to identifier renaming methods' change rates grow with the number of variable usages in the code. Since Vulnerability Prediction's source code is smaller than the two other tasks, this per-source code change rate is higher, on the contrary, when the program size grows, GRAPHCODEATTACK demonstrates better change rates.

> **RQ1 Conclusion:** GRAPHCODEATTACK outperforms CAR-ROT and ALERT in Authorship Attribution and Vulnerability Prediction tasks, with similar results in Clone Detection. For stealthiness, GRAPHCODEATTACK achieves reasonable change rates. GRAPHCODEATTACK gives a better code change rate on larger source codes while ALERT and CARROT's change rates grow with the length of the input code.

5.2 RQ2. What are the most effective patterns on each problem and model?

To understand which patterns contribute the most to the success of adversarial attacks, we evaluate the frequency with which a pattern is added in successful adversarial examples. We report the Top-3 most frequently occurring patterns for task and model combination. We do not count the dead code wrapper (e.g., if False) since they are not the original patterns. Since the tasks are done in different languages, we group the equivalent AST between different languages to count the patterns' frequency. For example, comparison_operator in Python between two expressions is equivalent to a binary_expression in C++ and Java, block in Python is equivalent to compound_statement in C++ and Java. If

Table 3. Code Change Rate of GraphCodeAttack, ALERT, and Carrot, μ_{TC} and σ_{TC} are the total number of tokens added and the standard deviation. μ_{TCR} and σ_{TCR} are the token change rate and the standard deviation respectively

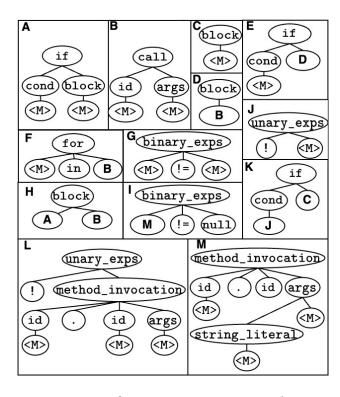| Method	Target Model	Authorship Attribution				Vulnerability Prediction				Clone Detection			
		μ_{TC}	σ_{TC}	μ_{TCR}	σ_{TCR}	μ_{TC}	σ_{TC}	μ_{TCR}	σ_{TCR}	μ_{TC}	σ_{TC}	μ_{TCR}	σ_{TCR}
GCA	CB	90.27	58.38	0.136	0.149	56.33	38.96	0.570	0.378	98.13	97.98	0.255	0.367
	GCB	112.833	64.4	0.1359	0.1724	62.06	47.165	0.463	0.794	40.75	33.75	0.03	0.04
ALERT	CB	151.95	144.254	0.1416	0.102	136.14	320.18	0.1295	0.086	69.68	97.191	0.112	0.0644
	GCB	325.94	195.98	0.344	0.124	159.302	371.39	0.121	0.086	153.36	193.34	0.264	0.1376
CARROT	CB	113.65	120.62	0.24	0.193	107.133	209.8	0.136	0.149	88.87	232	0.103	0.121
	GCB	129.101	161.29	0.26	0.18	112.61	179.61	0.215	0.2	170.95	297.866	0.2344	0.168

Figure 4. Top frequent patterns among attacks

Table 4. Top frequent patterns in a successful attack, CF means the patterns contain a control-flow element (e.g., `if`, `else-if`, `else`, `for`, `while`, etc.) and DF means the patterns contain calculations relating to variables (e.g., identifier nodes), L means the patterns contains Literals

Task	Model	Pattern ID	Pattern type	Frequency
Authorship Attribution	CB	A	CF	0.987
		G	DF	0.703
		B	CF, DF	0.604
	GCB	B	CF, DF	0.586
		D	DF	0.432
		F	CF, DF	0.211
Vulnerability Prediction	CB	E	CF	0.864
		K	CF	0.807
		I	CF	0.174
	GCB	B	CF, DF	0.413
		H	CF, DF	0.283
		E	CF, DF	0.249
Clone Detection	CB	G	DF	0.448
		L	CF, DF	0.085
		M	DF, L	0.023
	GCB	G	DF	0.615
		K	CF, DF	0.482
		E	CF, DF	0.448

an AST pattern only appears in a single language, we report the original pattern.

Table 3 and Figure 4 present the top frequently appearing patterns in successful attacks for each task and target model and Figure 4 depicts these patterns in detail. The results indicate that patterns' effectiveness varies across different tasks and target models. Particularly on all 3 tasks, CodeBERT has the 2 top frequently appeared patterns containing control-flow elements and 1 top pattern that contains data-flow elements. While GraphCodeBERT also has 2/3 top patterns containing control-flow elements, all of its top-frequented patterns in the successful attack contain data-flow elements. This hints that GraphCodeBERT models rely more on

variable and data-flow operations (which is consistent with the design of the model).

Among different tasks, Authorship Attribution and Clone Detection both have 4/6 patterns containing control flow and 5/6 patterns using data flow elements. Vulnerability Prediction has 6/6 patterns containing control flow elements and 3/6 containing data flow elements. This suggests that Authorship Attribution and Clone Detection have more reliance on data flow elements while the Vulnerability Prediction model leans more toward the control flow elements.

RQ2 Conclusion: Among the most frequently occurring patterns, CodeBERT frequently exhibits control-flow patterns, while GraphCodeBERT relies more on data-flow operations. Authorship Attribution and Clone Detection depend on data flow elements, whereas Vulnerability Prediction leans towards control flow elements.

5.3 RQ3. How effective is GraphCodeAttack in improving the robustness of code models?

Settings. To answer this question, we experiment with adversarial retraining, a process that involves fine-tuning the target model with adversarial examples to improve its robustness against potential attacks. To do this, we randomly insert the patterns generated by GraphCodeAttack into the training dataset as a data augmentation step. While training, on each data sample, we set the probability of applying a perturbation to 0.5. This probability indicates how likely we would apply a perturbation on each sample. Setting this probability to 0.5 balances the use of original and perturbed samples in the training process. Recall that GraphCodeAttack can insert dead code multiple times, we also set the maximum number of perturbations to 5, meaning that we will insert the dead code for the maximum of 5 times, which is half the number of greedy steps that GraphCodeAttack uses in attacking.

We obtain the ALERT [50]'s fine-tuned model from the official GitHub repository[4]. For CARROT [53], we follow the original procedure and obtain the perturbed samples that either change the model's prediction or drop the target model's confidence towards the original prediction, we augment the training set with these samples and fine-tune until the model converges.

We test the robustness of the fine-tuned models on the adversarial samples in RQ1 of both ALERT, CARROT, and GraphCodeAttack itself. The robustness measurement of a single adversarial sample is defined according to CARROT [53]: we measure the ratio of making correct predictions on the set of generated adversarial examples in RQ1.

Results. The results of adversarial training are presented in Table 5. Overall, on Authorship Attribution, using the Wilcoxon Rank Sum Test, we obtained a p-value of 0.031 for CARROT vs ALERT, 0.043 for CARROT vs GraphCodeAttack, and 0.893 between ALERT and GraphCodeAttack. The Cliff's Delta for the three pairs (CARROT vs ALERT, CARROT vs GraphCodeAttack and ALERT vs GraphCodeAttack) are -0.444 (medium), -0.472 (medium), and 0.028 (small) respectively. This demonstrates that ALERT and GraphCodeAttack exhibit similar performance in improving robustness, and both methods outperform CARROT. It is worth noting that for both Vulnerability Prediction and Clone Detection, the robustness improvements across different adversarial fine-tuning methods are similar for all baselines.

Interestingly, in the case of Authorship Attribution and Vulnerability Prediction, both CARROT and GraphCodeAttack's adversarial results are difficult to defend against. For CARROT, this can be attributed to the fact that changes in variables can be more pronounced than in ALERT, owing to the lack of natural constraints. For GraphCodeAttack, the difficulty in defense may be due to the

---
[4]https://github.com/soarsmu/attack-pretrain-models-of-code

variability of GraphCodeAttack's attacks: since the inserted attacks in GraphCodeAttack are filled using a pre-trained language model, the filled masks may vary, making it challenging to defend against the produced attacks.

Finally, while retraining with GraphCodeAttack and ALERT results in a similar defense, we emphasize that GraphCodeAttack only employs an augmentation method instead of performing a full adversarial attack on the target model or the target model's response itself, giving it finetuning an advantage of efficiency and in deployment.

RQ3 Conclusion: GraphCodeAttack's fine-tuning, although not requiring the target models' feedback nor a full adversarial attack on the training dataset like ALERT and CARROT, achieves similar performance with ALERT and better performance in comparison with CARROT.

6 THREATS TO VALIDITY

6.1 Threats to Internal Validity

The effectiveness of GraphCodeAttack relies on the mined AST patterns. If the pattern mining process is not comprehensive or biased towards specific patterns, it may affect the success rate of our adversarial attacks. We mitigate this threat by using a model-agnostic approach to mine discriminative patterns and validating our method against multiple datasets and model architectures. The process of inserting the AST patterns into the code and the selection of target models may introduce randomness, which can potentially influence the results. We address this issue by conducting experiments with 3 runs and reporting the average performance, ensuring the stability and reliability of our results.

6.2 Threats to External Validity

The generalizability of our findings might be limited by the choice of datasets, models, and evaluation metrics used in our experiments. To mitigate this threat, we employed widely-used tasks and datasets that are included in the CodeXGLUE benchmark, as well as the popular pre-trained language models for code: CodeBERT and GraphCodeBERT.

6.3 Threats to Construct Validity

The choice of evaluation metrics can impact the interpretation of our results. In this study, we used the Attack Success Rate (ASR) to measure the effectiveness of our method, which was widely used in the previous state-of-the-art papers [50, 53]. Moreover, we also adopt the same metric used to assess the robustness improvement from previous studies [50, 53].

7 RELATED WORK

This section provides an overview of the relevant studies in the field. We divide these related works into two categories: (1) pre-trained models of code and (2) potential threats to these models.

Table 5. Robustness improvement of the target model after adversarial fine-tuning: Each major column indicates from which method the adversarial examples are generated, and each minor column indicates from which method the model is adversarially fine-tuned.

Task	Model	CARROT			ALERT			GraphCodeAttack		
		CARROT	ALERT	GCA	CARROT	ALERT	GCA	CARROT	ALERT	GCA
Authorship	CodeBERT	0.2528	0.6781	0.3793	0.8095	0.8736	0.8095	0.2916	0.6067	0.7661
Attribution	GraphCodeBERT	0.00	0.0253	0.0253	0.5143	0.9621	0.9143	0.0096	0.027	0.4144
Vulnerability	CodeBERT	0.5145	0.5364	0.495	0.8518	0.8811	0.7362	0.5957	0.5786	0.6047
Prediction	GraphCodeBERT	0.5635	0.5242	0.5257	0.7966	0.8904	0.7893	0.578	0.582	0.5995
Clone	CodeBERT	0.9568	0.9606	0.9722	0.9012	0.9190	0.9120	0.9232	0.9341	0.9674
Detection	GraphCodeBERT	0.9434	0.9032	0.9322	0.9123	0.9104	0.9089	0.9233	0.9142	0.9258

7.1 Pre-trained Models of Code

Large language models, such as the BERT [11, 28] and GPT [7, 39] families, have achieved remarkable performance in various natural language processing tasks. This success inspired researchers to develop pre-trained models for programming languages to capture code semantics and improve code-related tasks.

The trend began with CodeBERT [13], based on RoBERTa [28], which was pre-trained on the bimodal CodeSearchNet dataset [18]. It has two pre-training objectives: masked language modeling and replaced token detection. GraphCodeBERT [16] further incorporates code graph structure, adding data flow edge prediction and node alignment. Other models, such as CuBERT [21] and C-BERT [9], focus on Python and C source code, respectively. These *encoder-only* models generate code embeddings for downstream tasks.

Another type of code model is *decoder-only* models, primarily focused on code generation tasks. The GPT-based code model is a well-known decoder-only architecture. Lu et al. introduce CodeGPT in the CodeXGLUE benchmark [29], utilizing the GPT-2 architecture and pre-trained on CodeSearchNet [18]. Larger models include InCoder [14] and CodeGen [32] with 16.1B parameters. OpenAI's Codex [10] powers Microsoft CoPilot. A recent study suggests that smaller models trained on high-quality datasets can outperform larger models [2].

Researchers have also applied *encoder-decoder* architecture to code models. Inspired by BART [25] and T5 [40], they propose models like PLBART [1] and CodeT5 [49], experimenting with pre-training tasks such as masked span prediction and masked identifier prediction. Other models include DeepDebug [12], Prophetnet-x [37], CoTexT [35], and SPT-Code [34]. These code models demonstrate remarkable performance on various code-related tasks, including code completion, code summarization, and code generation [33].

7.2 Threats to Code Models

Despite their impressive performance, code models remain vulnerable to various attacks. Understanding these vulnerabilities is crucial for enhancing their security and protecting their downstream applications.

One significant threat is the susceptibility of code models to adversarial examples [6]. Yefet et al. [52] were among the first to study

adversarial robustness in code models, employing FGSM [15] to rename variables and target code models like code2vec [3]. Jordan et al. [17] extend [52] work by considering additional transformations such as converting for loop to while loop. Srikant et al. [43] utilize PGD [30] to generate stronger adversarial examples. These studies assume *white-box* access to the code models, meaning that the attacker has access to the model's parameters and gradient information.

Attacks can also be conducted in a *black-box* manner. Zhang et al. [54] propose Metropolis-Hastings Modifier (MHM) to for black-box adversarial example generation. Rabin et al. [38] and Applis et al. [5] use semantic-preserving and metamorphic transformations, respectively, to assess code model robustness. Tian et al. [47] employ reinforcement learning for attacks, while Jia et al. [20] demonstrate that adversarial training improves code model robustness and correctness. Pour et al. [36] proposed leveraging renaming variables, argument, method, and API names as well as adding argument, print statement, for loop, if loop, and changing returned variables to generate adversarial source codes to improve code models' robustness. Yang et al. [50] emphasize the naturalness requirement for code model adversarial examples and develop ALERT, which uses genetic algorithms for example generation. Zhang et al. [53] propose CARROT, which employs worst-case performance approximation to measure code model robustness. Both ALERT and CARROT demonstrate state-of-the-art performance. GraphCodeAttack focuses on crafting more complex adversarial examples rather than emphasizing naturalness.

Researchers have also studied threats like data poisoning and backdoor attacks. Ramakrishnan et al. [41] propose fixed and grammar triggers to insert backdoors into code models, while Wan et al. [48] inject similar triggers into code search models. Yang et al. [51] use adversarial examples for stealthy backdoor injection, and Li et al. [26] generate dynamic triggers using language models, proposing an effective defense method. Schuster et al. [42] conduct data poisoning for insecure API usage, while Nguyen et al. [31] assess the risk of malicious code injection in API recommender systems. Sun et al. [44] demonstrate data poisoning can protect open-source code from unauthorized training.

8 CONCLUSION AND FUTURE WORK

We presented GraphCodeAttack, an adversarial attack tool for pre-trained code models to better evaluate the robustness of code models. We evaluate the robustness of two popular code models (e.g., CodeBERT and GraphCodeBERT) against our proposed approach on three tasks: Authorship Attribution, Vulnerability Prediction, and Clone Detection. The experimental results suggest that our proposed approach significantly outperforms state-of-the-art approaches in attacking code models such as CARROT and ALERT. Based on the average attack success rate (ASR), GraphCodeAttack achieved 30% improvement over CARROT and 33% improvement over ALERT respectively. We also evaluate the produced attack quality with the usage of code change rate and shows that Graph-CodeAttack produces successful attacks with fewer token change in general and the code change rate decrease with larger code files. Furthermore, GraphCodeAttack's adversarial fine-tuning has a similar performance with ALERT in enhancing model robustness, while requiring neither the target model's output nor conducting a full adversarial attack on the training data.

For future work, we plan to investigate the impact of different perturbations to improve GraphCodeAttack's performance. Additionally, we aim to explore more attack scenarios, such as multi-label and multi-class settings, to further evaluate the effectiveness of GraphCodeAttack and enhance its generalizability across a wider range of code-related tasks.

REFERENCES

[1] Wasi 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*. Association for Computational Linguistics, Online, 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, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! arXiv:2301.03988 [cs.SE]

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[4] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 65–82.

[5] 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)*. 1377–1381. https://doi.org/10.1109/ASE51524.2021.9678706

[6] Pavol Bielik and Martin Vechev. 2020. Adversarial Robustness for Code. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 896–907. https://proceedings.mlr.press/v119/bielik20a.html

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901.

[8] Nghi D. Q. Bui, Yue Wang, and Steven Hoi. 2022. Detect-Localize-Repair: A Unified Framework for Learning to Debug with CodeT5. arXiv:2211.14875 [cs.SE]

[9] Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. *CoRR* abs/2006.12641 (2020). arXiv:2006.12641 https://arxiv.org/abs/2006.12641

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[11] 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, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[12] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating Bug-Fixes Using Pretrained Transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3460945.3464951

[13] 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*. Association for Computational Linguistics, 1536–1547.

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=hQwb-lbM6EL

[15] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*. http://arxiv.org/abs/1412.6572

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu andz 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*.

[17] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, 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)*. 526–537. https://doi.org/10.1109/SANER53432.2022.00070

[18] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[19] Akshita Jha and Chandan K. Reddy. 2023. CodeAttack: Code-Based Adversarial Attacks for Pre-trained Programming Language Models. arXiv:2206.00052 [cs.CL]

[20] Jinghan Jia, Shashank Srikant, Tamara Mitrovska, Chuang Gan, Shiyu Chang, Sijia Liu, and Una-May O'Reilly. 2023. CLAWSAT: Towards Both Robust and Accurate Code Models. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

[21] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*. PMLR, 5110–5121.

[22] Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code? arXiv:2108.11308 [cs.SE]

[23] Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. 2019. CTRL: A Conditional Transformer Language Model for Controllable Generation. arXiv:1909.05858 [cs.CL]

[24] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2022. Taxonomy of Attacks on Open-Source Software Supply Chains. arXiv:2204.04008 [cs.CR]

[25] 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*. Association for Computational Linguistics, Online, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

[26] Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2022. Poison Attack and Defense on Deep Source Code Processing Models. *arXiv preprint arXiv:2210.17029* (2022).

[27] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. arXiv:2012.14631 [cs.SE]

[28] 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

[29] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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. arXiv:2102.04664 [cs.SE]

[30] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[31] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. 2021. Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 253–265. https://doi.org/10.1109/ASE51524.2021.9678946

[32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.

[33] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. arXiv:2302.04026 [cs.SE]

[34] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2006–2018. https://doi.org/10.1145/3510003.3510096

[35] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Association for Computational Linguistics, Online, 40–47. https://doi.org/10.18653/v1/2021.nlp4prog-1.5

[36] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. 2021. A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding. arXiv:2101.07910 [cs.SE]

[37] Weizhen Qi, Yeyun Gong, Yu Yan, Can Xu, Bolun Yao, Bartuer Zhou, Biao Cheng, Daxin Jiang, Jiusheng Chen, Ruofei Zhang, Houqiang Li, and Nan Duan. 2021. ProphetNet-X: Large-Scale Pre-training Models for English, Chinese, Multi-lingual, Dialog, and Code Generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 232–239. https://doi.org/10.18653/v1/2021.acl-demo.28

[38] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.

[39] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[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. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[41] G. Ramakrishnan and A. Albarghouthi. 2022. Backdoors in Neural Models of Source Code. In *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 2892–2899. https://doi.org/10.1109/ICPR56361.2022.9956690

[42] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1559–1575.

[43] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating Adversarial Computer Programs using Optimized Obfuscations. *ICLR* 16 (2021), 209–226.

[44] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. CoProtector: Protect Open-Source Code against Unauthorized Training Usage with Data Poisoning. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 652–660. https://doi.org/10.1145/3485447.3512225

[45] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. https://doi.org/10.1109/icsme.2014.77

[46] Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans-Peter Kriegel, Alex Smola, Le Song, Philip S. Yu, Xifeng Yan, and Karsten M. Borgwardt. 2010. Discriminative frequent subgraph mining with optimality guarantees. *Statistical Analysis and Data Mining* 3, 5 (Aug. 2010), 302–318. https://doi.org/10.1002/sam.10084

[47] Junfeng Tian, Chenxin Wang, Zhen Li, and Yu Wen. 2021. Generating Adversarial Examples of Source Code Classification Models via Q-Learning-Based Markov Decision Process. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 807–818. https://doi.org/10.1109/QRS54544.2021.00090

[48] Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. You See What I Want You to See: Poisoning Vulnerabilities in Neural Code Search. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1233–1245. https://doi.org/10.1145/3540250.3549153

[49] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 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, EMNLP 2021*.

[50] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*. ACM. https://doi.org/10.1145/3510003.3510146

[51] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2023. Stealthy Backdoor Attack for Code Models. https://doi.org/10.48550/ARXIV.2301.02496

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

[53] 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 Trans. Softw. Eng. Methodol.* 31, 3, Article 50 (apr 2022), 40 pages. https://doi.org/10.1145/3511887

[54] 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. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 01 (Apr. 2020), 1169–1176.

[55] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. arXiv:2302.05527 [cs.SE]

[56] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. arXiv:1909.03496 [cs.SE]