ALANCA: Active Learning Guided Adversarial Attacks for Code Comprehension on Diverse Pre-trained and Large Language Models

***Abstract*—Neural code models have demonstrated their efficacy across a range of code comprehension tasks, including vulnerability detection, code classification, automatic code summarization, completion, clone detection, etc. Yet, a substantial gap exists in our understanding of the robustness of models in the realm of code comprehension and its associated applications. To probe and illuminate the robustness of code, recent efforts have sought to employ NLP-like techniques to craft adversarial code instances, primarily by perturbing variable and token names. It's worth noting that the semantics of source code predominantly surface through its structural elements, such as abstract syntax trees and control flow graphs, which fundamentally differ from natural languages. The question remains open: Can we perturb the structural aspects of code while preserving its semantics, thereby generating more disruptive adversarial examples that elude current structural-unaware approaches? Moreover, orchestrating adaptive adversarial attacks on diverse neural code models with varying architectures poses formidable challenges, especially in real-world scenarios characterized by constraints on target model access and querying.**

In this paper, we introduce *ALANCA*, an active-learning guided adversarial attack framework tailored for neural code models. Leveraging semantic-preserving translations, combined with an adaptive adversarial discriminator and token selector, *ALANCA* excels in executing adversarial attacks with high success rates, exceptional generation quality, and adaptability across different target models. We substantiate *ALANCA*'s efficacy through comprehensive evaluations across four distinct code comprehension tasks, demonstrating its ability to effectively confound a range of neural models, including pre-trained models and LLMs used in software engineering.

I. Introduction

The demand for intelligent programming has surged, driven by the expanding complexity of software development requirements. Intelligent programming is pivotal for ensuring software security, program quality, and development efficiency. In recent years, neural code models, including pre-trained code models and large language models, have showcased remarkable performance across various code-related tasks, particularly in code comprehension. These tasks encompass vulnerability detection [1]–[3], code classification [4], [5], code summarization [6]–[8], code completion [9], [10], and clone detection [11], [12]. While neural code comprehension models hold the promise of augmenting programming intelligence, they also raise concerns related to vulnerability and lack of robustness. Despite their impressive performance, the issue of vulnerability in code models remains largely unresolved. With the growing adoption of AI-assisted development tools and defect-detecting models, it is imperative to prioritize the matter of robustness in neural code models and prevent the inadvertent introduction of faulty code or vulnerabilities into real-world systems.

Adversarial attacks pose significant threats to non-robust neural models across diverse domains. Notably, techniques like the Fast Gradient Sign Method (FGSM) [13] illustrate how even minor adjustments to ordinary inputs, imperceptible to humans, can lead to neural models producing misleading results. In the domain of code, adversarial attacks primarily revolve around text modification [14]–[17], code refactoring [18]–[22] and other techniques like obfuscation [23]. However, the precise threshold for perturbations and the criteria for defining an adversarial example remain ambiguous. This lack of standardized guidelines results in alterations to code semantics, leading to changes in program functionality, including variable assignments and control flow adjustments. Sometimes, these modifications become glaringly apparent and fail manual inspection, either due to excessive text changes or conspicuous renaming.

In the context of adversarial attacks on code models, the majority of existing studies operate within a black-box setting. This implies that attackers lack access to internal information about the target models, including their structure, parameters, and gradients. Only a select few works, such as those discussed in [22], [24], have delved into white-box attack settings where attackers can utilize gradient information for precision-targeted attacks. However, it's crucial to acknowledge that real-world scenarios impose even more stringent limitations on adversarial attacks. Many code model applications come as packaged software or offer restricted APIs, limiting the frequency of queries to the target model. Unfortunately, these constraints are frequently overlooked or inadequately considered in existing research.

Furthermore, considering the diversity in types, architectures, and scales of neural code models, as well as the variations among different code comprehension tasks, developing universal adversarial attack tools capable of effectively and efficiently targeting all models and tasks is a formidable challenge.

To tackle these challenges and limitations, we introduce *ALANCA*, an active-learning guided adversarial attack framework designed to execute effective black-box attacks on a wide array of code comprehension models and tasks. In our approach, we introduce a guided code transformer that employs a range of semantic-preserving code transformations

guided by statistical information to initially generate a pool of candidate adversarial examples. To adaptively identify the most effective adversarial examples from this pool, we utilize a learnable adversarial discriminator and a token selector. The discriminator aids in pinpointing the adversarial examples most likely to deceive the target model, while the token selector concentrates on refining and enhancing the selected examples. We integrate active learning into the proposed adversarial attack framework, enabling the efficient training of *ALANCA* within the strict black-box setting.

To demonstrate the effectiveness and efficiency of our approach, we conducted extensive evaluations on four distinct code comprehension tasks: Code Summarization, Method Name Prediction, Code Classification, and Code Clone Detection. We tested *ALANCA* on eight different models, including both pre-trained models and LLMs in SE. The results of our evaluations clearly demonstrate that *ALANCA* effectively confuses and neutralizes various neural models with high efficiency.

We summarize our main contributions as follows:

- We propose *ALANCA*, an active-learning guided adversarial attack framework for neural code models, which is adaptive to almost all of the code comprehension tasks. To the best of our knowledge, this is the first framework that enables effective and scalable adversarial attacks on various neural code models, widely ranging from small Seq2Seq-based models to large language models (LLMs).
- Through an extensive evaluation, we demonstrate the effectiveness and efficiency of *ALANCA* compared to existing adversarial attack methods on code. Our evaluation involves eight target models and four code comprehension tasks, and *ALANCA* consistently achieves a high attack success rate.
- We have made the implementation of our approach and the datasets openly available to facilitate further research on the robustness of neural code models and other software engineering practices.

II. Background and Preliminaries

In this section, we introduce the preliminary concepts related to the adversarial attack on code models and the background knowledge about techniques in our proposed attack framework, *ALANCA*.

A. Code Comprehension Neural Models

Code comprehension problems encompass a variety of software engineering problems. They can be generally defined as transformations $\mathcal{F} = f : \mathcal{X} \rightarrow \mathcal{Y}$, where code snippets ($\mathcal{X}$) are transformed into desired representations ($\mathcal{Y}$). These problems can be categorized into code-to-text problems (such as code summarization and method name prediction) and code-code problems (including code classification, clone detection, and code repair). The inputs ($\mathcal{X}$) for code comprehension tasks can vary from text sequences at the character or token level to different types of code abstraction models, such as abstract parsing trees, control flow graphs, value flow graphs, and code property graphs.

In recent years, neural models, including pre-trained models [7], [11], [12], [25], [26] and Language Models (LLMs) [27]–[29], have been increasingly applied to enhance code comprehension. These models are trained under supervision with labeled datasets $\mathcal{D} = (\mathcal{X}, \mathcal{Y}_g)$, where $\mathcal{Y}$ represents the ground-truth labels for code snippets. Neural models extract features from different types of input code snippets and encode them into code representation vectors using various network layers. Subsequently, these code representation vectors are decoded into different output formats based on the specific tasks at hand.

B. Adversarial Robustness of Code Neural Models

In the context of generating adversarial examples for code comprehension models, e.g. a code classifier model $f_c : \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ represents the set of all input code snippets. The objective of an adversarial attack is to transform an input example to mislead the classifier. For a given code snippet x in the input set $\mathcal{X}$, let x^{adv} denote an adversarial example of x, such that the distance between the adversarial example and the original input is smaller than a threshold ϵ. Formally, we have:

$$f_c(x^{adv}) \neq f_c(x), \text{ s.t. } ||x^{adv} - x|| < \epsilon,$$

The process of finding an adversarial example can be formulated as a constrained optimization problem:

$$x^{adv} = \underset{||x'-x||<\epsilon}{\arg\max} (g_c(x', l)),$$

where $g_c(x', l)$ is a metric that quantifies the adversarial effect on the classifier. Prior research on white-box attacks has introduced gradient-based methods, such as FGSM [13], PGD [30], and CW [31], as solutions to the optimization problem. These methods rely on gradients obtained from the target model to introduce perturbations within a specified radius (ϵ) to create the most effective adversaries. However, in real-world scenarios, acquiring the necessary information for conducting white-box attacks, including knowledge about the architecture, parameters, and loss of the classifier, can be challenging. A more common and practical scenario is the black-box setting, where only the input features and output results of the target model are accessible, and the number of queries is limited.

Moreover in the domain of code, as discussed in [24], the process of generating adversarial examples in discrete domains (natural language or code), substantially differs from that in the image domain, where adversarial examples have been extensively researched [32]. In discrete domains, quantifying the distance between two examples is a more intricate task, and determining a suitable threshold ϵ for defining permissible perturbations is also a challenging endeavor. The concept of dual-channel semantics for code, introduced by [33], offers a promising approach to addressing these challenges. This approach combines a precise, formal representation of functionality with a natural and human-understandable representation of language, presenting a potential solution for these issues.

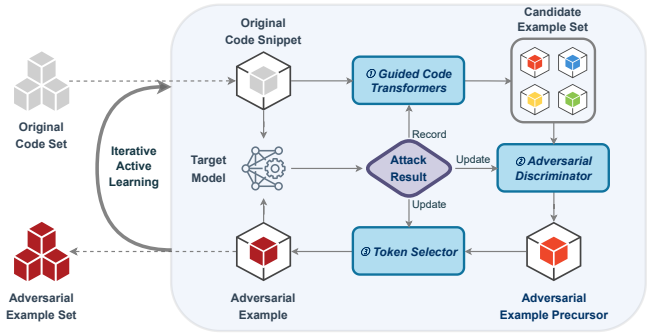

Fig. 1. Overview of *ALANCA*

C. Active Learning

In the problem of supervised learning where labeled data are limited or expensive to acquire, active learning (AL) provides an effective solution. Pool-based active learning is a common approach with a model $y = h(x)$ to train and an initial pool of unlabeled data $\mathcal{D}_{\mathcal{U}}$. Through a limited number of query iterations, we can actively select certain samples from the unlabeled pool to annotate and add to the labeled data set $\mathcal{D}_{\mathcal{L}}$. During the active learning process, we gradually improve the rationality of sampling by performing iterations. In each iteration, we sample a subset of unlabeled data, denoted as ${\mathcal{D}_{\mathcal{U}}}^n_{-k_n}$, and query an oracle to obtain their labels. These newly labeled examples are then added to the labeled data set $\mathcal{D}_{\mathcal{L}}$, resulting in an updated set ${\mathcal{D}_{\mathcal{L}}}^{n+1} = {\mathcal{D}_{\mathcal{L}}}^n \cup \{(x,l)|x \in {\mathcal{D}_{\mathcal{U}}}^n_{-k_n}\}$. Simultaneously, we remove the queried samples from the unlabeled pool, leading to an updated pool ${\mathcal{D}_{\mathcal{U}}}^{n+1} = {\mathcal{D}_{\mathcal{U}}}^n - {\mathcal{D}_{\mathcal{U}}}^n_{-k_n}$.

Each iteration uses a sampling model to decide which examples to query from the unlabeled pool. This sampling model can be a heuristic algorithm or a learnable model. Additionally, after obtaining the labels for the sampled examples, the sampling model itself is updated using the new labeled set $\{(x,l)|x \in {\mathcal{D}_{\mathcal{U}}}^n_{-k_n}\}$. The total number of queries is limited by a preset number, where the sum of queries in all iterations $\sum_{0<n\leq N} k_n \leq q$ should not exceed q. The active learning system continues to iterate until the model performance meets the desired requirements or until the number of queries is exhausted.

III. Attack Methodology of *ALANCA*

A. Overview

In order to effectively and efficiently perform adversarial attacks on code neural models, especially within strict black-box settings, we present an active learning-guided adversarial attack framework for code comprehension models, named *ALANCA* in short. An overview of *ALANCA* is presented in Figure 1. *ALANCA* comprises three key components. All the components in *ALANCA* are learnable and designed to be adaptive to different types of target models and downstream code comprehension tasks, e.g., code summarization, classification, etc. Conducting adversarial attacks within a stringent black-box setting poses challenges in training the learnable components, primarily due to restricted access to the target model. To overcome this issue, we integrate active learning into the proposed adversarial attack framework, enabling the training of *ALANCA* with only a limited number of labels obtained from the target model.

Algorithm 1 Active Adversarial Attacking Algorithm

Input: Victim Model V;
Code Snippet Dataset $\mathcal{D} = \{(x,y) \mid x \in \mathcal{X}, y \in \mathcal{Y}\}$;
Guided Code Transformer $\mathbb{T}$;
Adversarial Example Discriminator M_d
Token Selector M_{ts}
Max Query Number to Victim Model K_q
Output: Adversarial Example Set $\mathcal{X}^{adv}$ from the attacking algorithm $\mathbb{A}((x,y);V)$

Initialization: $\mathcal{X}^{adv} \leftarrow \varnothing$
repeat
  // Guided code transformation
  for $(x,y) \in \mathcal{D}$ **do**
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{x^t = \mathbb{T}(x)\}$
  end for
  // Rank generated examples
  $\mathcal{S} \leftarrow \varnothing$
  for $x^t \in \mathcal{T}$ **do**
    $\mathcal{S} \leftarrow \mathcal{S}\ \{(M_d(x^t,x), x^t \mid x^t = \mathbb{T}(x))\}$
  end for
  $\mathcal{T}_d \leftarrow \{x^t_i \mid (s_i, x^t_i) \in \mathcal{S},\ i \in \arg\max\mathrm{K}\{s_i \mid \forall i,\ (s_i, x^t_i \in \mathcal{S})\}\}$
  // Substitute [T] in generated examples
  $\mathcal{X}^{adv} \leftarrow \varnothing$
  for $x^t \in \mathcal{T}_d$ **do**
    $\mathcal{X}^{adv} \leftarrow \mathcal{X}^{adv} \cup \{M_{ts}(x^t,x) \mid x \in \mathcal{X},\ x^t = \mathbb{T}(x)\}$
  end for
  // Update parameters of $\mathbb{T}$, M_d and M_{ts}
  $\mathcal{Y}^{adv} = V(\mathcal{X}^{adv})$
  Record the attack result in $\mathbb{T}$ to update the guiding parameters.
  $l_d = L_d(\mathcal{Y}^{adv}, \mathcal{Y})$
  $l_{ts} = L_{ts}(\mathcal{Y}^{adv}, \mathcal{Y}, \mathcal{X}^{adv}, \mathcal{X})$
  Update M_d and M_{ts} by backpropagating l_d and l_{ts}.

  $K_q \leftarrow K - 2 * K$
until $K_q < 2 * K$

B. Active-learning Guidance in Adversarial Attack

Within *ALANCA*, we employ an iterative active learning algorithm to conduct effective attacks while working with limited knowledge about the target model and a restricted number of queries, as detailed in Algorithm 1. Each component of *ALANCA* assumes a pivotal role in every iteration of the attack process, contributing significantly to the overall effectiveness and efficiency of the adversarial attack:

Component ❶: Guided Code Transformer § III-C The initial step in *ALANCA* involves generating candidate adversarial examples through the application of various code transfor-

mation methods (Lines 3 to 6). These transformations are carefully designed to maintain the semantic meaning of the code while making it challenging for the target model to correctly understand and classify. To ensure that the generated adversarial examples maintain their semantic integrity while remaining imperceptible to humans, *ALANCA* employs a series of AST-based mutation techniques. During this phase, *ALANCA* produces a varied collection of candidate adversarial examples, denoted as $\mathcal{T}$, each meticulously crafted to explore various attack strategies.

Component ❷: Adversarial Example Discriminator § III-D *ALANCA* identifies robust vulnerabilities in code snippets and their corresponding non-robust types specific to each target model and task. In practical implementation, as demonstrated in Lines 7 to 12 of Algorithm 1, *ALANCA* employs a discriminator module to score and rank candidate examples in the set $\mathcal{T}$. The discriminator module is trained to gauge the quality and effectiveness of the generated adversarial examples. The top-K examples, deemed most likely to be effective in an attack, are chosen as the precursors of the desired adversarial example $\mathcal{T}_d$.

Component ❸: Token Selector § III-E In the context of *ALANCA*, certain code transformations, such as identifier renaming and code insertion, may necessitate the introduction of new words or tokens into the resulting code. These newly introduced elements are initially denoted as "[T]" and are earmarked for substitution within the candidate example set $\mathcal{T}_d$. To ensure the quality of the generated adversarial examples in terms of both attack effectiveness and human imperceptibility, *ALANCA* incorporates the token selector component. The token selector is specifically designed to choose the most suitable tokens for substitution in the "[T]" parts of the generated adversarial examples, as indicated in Lines 13 to 17 of Algorithm 1.

In the proposed active learning-guided attack framework, the process of selecting examples to query from the target model and updating the components (Lines 19 to 23) is carefully crafted to capture essential information necessary for conducting effective adversarial attacks. A crucial element of this selection process involves harnessing the structural similarities between the target model and the learnable components. By identifying and querying examples with the target model, *ALANCA* gains valuable insights into the vulnerabilities and weaknesses of the target model. This, in turn, enables the learnable components to better emulate and exploit these structural commonalities in the generation of adversarial examples.

Furthermore, in the pursuit of refining the active-learning guided training process, *ALANCA* integrates several techniques, particularly when working with a restricted query budget within the strict black-box attack setting. These techniques, including multi-task loss functions and compressed vocabulary, will be explored in greater detail in the subsequent sections.

C. Statistics Guided Code Transformer

ALANCA first performs guided transformations on code snippets to generate the set of candidate adversaries $\mathcal{T}$. To provide a concrete definition of this generation process, we assume that $\mathbb{T} = \{\delta | \delta(x) = x'\}$ represents a set of code transformation functions, where $x \in \mathcal{O} \wedge x' \in \mathcal{T}$. According to the definition of adversarial examples in § II-B, all adversarial examples generated should ensure that the degree of perturbation remains within a specified threshold. This can be expressed as:

$$||x' - x|| = ||\delta(x) - x|| \leq \epsilon, \text{ s.t. } \delta \in \mathbb{T}, x \in \mathcal{O}, x' \in \mathcal{T}.$$

To generate code transformations of high quality, it is essential to consider the similarity or distance ($||x' - x||$) between two code snippets from two perspectives: (a) functional similarity and (b) inspectability for humans. Functional similarity refers to the equivalence in behavior between two pieces of code. Inspectability for humans refers to the ability of developers or reviewers to understand and analyze the differences between two code snippets.

To achieve this goal, we have proposed eight types of code transformation that introduce minimal structural perturbations and textual changes. Detailed descriptions and examples are presented in Table I. These transformations draw inspiration from the code refactoring and representation-changing processes observed during compilation. Refactoring involves making small modifications to the source code that preserve the program's behavior. Various techniques inspired by refactoring and compilation optimization have been proposed, e.g. symbol renaming and code reordering are common code transformations employed in refactoring practices [24], [34]. Additionally, some transformations are derived from optimization passes in the compilation process, such as dead code elimination $\rightarrow$ dead code insertion. In the design of *ALANCA*, we have chosen and developed these specific types of transformations meticulously. Our objective is to ensure a diverse range of refactoring techniques while also constraining the extent of textual changes introduced.

Several factors can significantly impact the effectiveness of the generated adversarial examples, e.g. the position where the code is modified or added, or the length of the newly added variable's name. To control this transformation position and guide the attack, *ALANCA* utilizes a series of parameters that are calculated based on statistics to determine where the modifications or additions occur within the code, listed in Table I. We use Spoon to travel across the abstract syntax trees (ASTs) of code snippets, identify possible transformation spots, and perform these semantic-preserving transformations, then convert the transformed ASTs back to code as candidate examples. During the adversarial example generation process, *ALANCA* iteratively transforms the code to generate different versions of the example. The attacking effectiveness of these generated examples is recorded by the code snippet transformer, which tracks the impact of the modifications on the adversarial behavior. Based on the effectiveness recorded,

TABLE I

DESCRIPTION AND EXAMPLES OF CODE TRANSFORMATIONS. GREEN STANDS FOR ADDED TOKENS, AND BLUE STANDS FOR MODIFIED TOKENS. IN THE GUIDING PARAMETERS COLUMN, p_i MEANS THE RELATIVE POSITION OF TRANSFORMATION IN A CODE SNIPPET, p_{st} MEANS THE INDEX OF SUBSTITUTED SUB-TOKEN IN AN IDENTIFIER, l_{st} AND l_b MEANS THE LENGTH OF THE ADDED IDENTIFIER, PRINTED WORDS, AND WRAPPED BLOCK.

Name	Functionality Description	Example	Guiding Parameters	Token Selection
Local Variable Renaming	rename a local variable by replacing a sub-token	int textLen = 0; ⇒ int [T]Len = 0;	p_i, p_{st}	✓
Parameter Renaming	rename a function parameter by replacing a sub-token	void f(String[] s, int len) ⇒ void f(String[] s, int [T])	p_i, p_{st}	✓
Variable Declaration Insertion	insert dead code of an unused variable declaration	int [T][T] = 1;	p_i, l_{st}	✓
Printing Insertion	insert a `System.out.println(s);` printing code line	System.out.println("[T] [T]");	p_i, l_{st}	✓
Do-While(false) Insertion	wrap a code block with a do-while(false) loop structure	int a = 0; ⇒ do{ int a = 0; }while(false);	p_i, l_b	
Statement Reordering	reorder two adjacent code statements with no variable def-use dependency	int a = 0; boolean b = false; ⇒ boolean b = false; int a = 1;	/	
Loop Converting	convert the type of loop between "for" and "while"	for(int i=0;i<n;++i){...} ⇒ {int i=0; while(i<n){... ++i;}}	/	
Equation Operators Exchanging	exchange the left and right side expressions of the "==" equation	a == b ⇒ b == a	/	

ALANCA updates the relevant position parameters, allowing *ALANCA* to adapt and refine its transformation strategy based on the observed performance of the adversarial examples generated. In the output stage, the code transformation component of *ALANCA* generates examples for each input code snippet based on the parameters associated with all possible refactor types. Within this component, any subtokens that require substitution or tokens that need to be generated are marked as "[T]".

D. Adversarial Example Discriminator

Different types of pre-trained code models with varying inner structures and input formats may interpret and represent code in distinct ways. This variation can result in biases in the vulnerabilities they exhibit regarding robustness. For example, CODEBERT was built from the pre-trained language model BERT with 125M parameters, while CODE2VEC has an encoder-only model with encoded paths of ASTs as input. These models can differ in terms of abstraction levels, granularity, and inherent biases in their code representations. Consequently, certain vulnerabilities or adversarial examples may be more effective or evident in one model compared to another. In addition, most code models are fine-tuned to perform different code-to-text or code-to-code downstream tasks. On the other hand, different code comprehension tasks place varying emphasis on different aspects of code input, e.g. in the code summaries task, the token-level semantic meaning of code may be more important than syntax-level structure information, while in the clone detection task, the situation may be just the opposite. In summary, when it comes to performing adversarial attacks, it is crucial to employ adaptive strategies that take into account the specific target models and tasks involved. This adaptability is especially important when dealing with automated black-box attack frameworks.

To address this challenge, we have incorporated a learnable discriminator module within *ALANCA*. This module is designed to identify vulnerable spots in code snippets and classify them according to their vulnerable type. For every original code snippet and its candidate adversarial examples, the discriminator predicted how the examples change the target model's prediction with a Siamese network structure. Then *ALANCA* ranks all of the candidates by the predicted output changing. Since the whole attack process is performed under the black-box setting, the query number for the target model is limited. This poses a challenge when it comes to gathering a large labeled training set for the discriminator. To overcome this challenge, we leverage existing code neural models and keep most of the parameters fixed in their encoders to keep the trainable part of the module lightweight. To leverage limited labeled examples obtained from queries during training iterations, we introduce two tasks for training the discriminator: regression of the prediction changing score s_{ij} and classification of the prediction changing type y_{ij}. The joint loss function for these tasks is defined as:

$$L_d = w_r(s_{ij} - h_{reg}(p'_{ij}, p_i))^2 - w_c \sum_{c=0,1,2} y_{ij} \log P(h_{cls}(p'_{ij}, p_i) = c),$$

Here, y_{ij} represents the three possible relationships between predictions f_c on p_i and p'_{ij}: prediction changes, prediction remains the same with a probability increase or decrease, or prediction remains the same with unchanged probability. The loss function consists of two parts: L_{reg} and L_{cls}, weighted by w_r and w_c, respectively, to balance their magnitudes.

E. Token Selector

As detailed in the output of the code transformation component, as discussed in § III-C, certain generated and selected adversarial examples may still contain "[T]" markers, signifying tokens that require substitution. To tackle this issue, we introduce a token selector module at this stage. The primary responsibility of this module is to forecast the most suitable choices for substituting the masked tokens.

There are two types of "[T]" marks that need to be replaced in this stage: (a) Sub-tokens within an identifier that require substitution for renaming purposes. (b) Newly added dead code. For the scenario of renaming identifiers, we introduce a constraint in the form of a penalty term within the training loss. This constraint aims to prevent excessive changes in the meaning of the identifiers' names after renaming, ensuring that the transformation remains contextually appropriate and not easy to notice in human reviewing. To quantify the penalty term for the renaming scenario, we measure the distance between the selected token and the original one. This distance is calculated using the cosine distance between the word vectors, which are embedded using a word2vec model. The loss function in training the token selector can be defined as:

$$\begin{aligned} L_{ts} = & \ w_r(s_{ij} - h_{reg}(p'_{ij}, p_i))^2 \\ & - w_c \sum_{c=0,1,2} y_{ij} \log P(h_{cls}(p'_{ij}, p_i) = c) \\ & - w_p D_{cos}(E_{w2v}(t'_{ij}), E_{w2v}(t_i)) \end{aligned}$$

In this context, t_i and t'_{ij} represent the original token and the token to be selected, E_{w2v} represents the work2vec embedding model, D_{cos} is the cosine distance measurement function. The penalty coefficient is denoted as w_p.

We employ CodeBERT [11] as the underlying framework for the token selector model. Similar to the discriminator module, the token selector model is constrained by the size of the training set. To address this limitation, we control the size of the candidate set and utilize a word vocabulary comprising the 1,500 most frequent tokens in identifiers. Besides, we keep the parameters of the CodeBERT encoder fixed during the fine-tuning of the token selector model for the sake of consistency and to harness the existing knowledge embedded in CodeBERT.

IV. Evaluation

In this section, we thoroughly evaluate the effectiveness and performance of *ALANCA*. We address the following three research questions to demonstrate the results of the proposed adversarial attack framework from different perspectives.

RQ 1 ***(Attack Effectiveness):*** Does our proposed adversarial attack framework (*ALANCA*) demonstrate sufficient effectiveness against code comprehension models?

RQ 2 ***(Attack Efficiency):*** Will *ALANCA* maintain its effectiveness under query-limited black-box attack settings, i.e., how is the efficiency of the *ALANCA* adversarial attack framework?

TABLE II
STATISTICS OF DATASETS

Target Dataset	Task	#Code Snippet	#Vocab	Supported Models
CodeSearchNet	① CS	324,439	50,265	except for AST-based models
Java-med	② MNP	382,986	60,000	all models
JAVA250	③ CC	73,143	8,000	all models
BigCloneBench	④ CD	9,126	2,800	except for LLMs

RQ 3 ***(Ablation Study):*** How effective are the different components in our adversarial attack frameworks, i.e., how is the effectiveness of each component over the baseline?

A. Experiment Setups

Downstream Tasks and Datasets. To conduct a thorough evaluation of our proposed adversarial attack framework *ALANCA*, we performed experiments on four distinct code comprehension tasks: ① Code Summarization, ② Method Name Prediction, ③ Code Classification, and ④ Code Clone Detection. These tasks encompass both code-to-text (① and ②) and code-to-code (③ and ④) scenarios. Table II shows some statistics of the datasets we used in each task.

For the ① Code Summarization task, we employed the *CodeSearshNet* [35] dataset, which includes code snippets from various sources along with associated comments. For the ② Method Name Prediction task, we utilized the *Java-med* dataset [8], which consists of top-starred Java projects from GitHub. The dataset for the ③ Code Classification task was the *JAVA250* dataset from *CodeNet* [36], comprising Java programs submitted to OJ platforms. The code snippets were labeled according to the programming question number. For the ④ Clone Detection task, we used the *BigCloneBench* dataset, which contains pairs of Java methods along with labels indicating whether the methods are semantically equivalent.

To ensure that simple methods were not affected by significant semantic changes during the code transformation stage of our attack, we implemented a filtering process. We excluded code snippets that lacked function parameters, variable declarations, and methods containing fewer than five lines before conducting our experiments. Additionally, we assigned subtokens to each code example, excluding non-meaningful tokens like articles or single-character tokens, specifically within identifiers. This labeling facilitated subsequent substitution or insertion procedures.

Target Models. As mentioned in § II-A, neural models for code comprehension can vary significantly in terms of their model structure and input format. In this paper, our focus is primarily on investigating the robustness of three main types of code neural models:

(a) AST-based models, including CODE2VEC [7] and CODE2SEQ [8].

(b) Pre-trained transformer models, including CODEBERT [11], GRAPHCODEBERT [26], PLBART [37], and CODET5 [37].

(c) LLMs, including CHATGPT [38] and CHATGLM 2 [39].

CODE2VEC [7] is a neural model that represents a code snippet using a set of paths extracted from its Abstract Syntax Tree (AST). Each path consists of terminal nodes and types of passing nodes, which are then encoded into a vector using multilayer perceptrons (MLPs). To highlight the significance of each path, CODE2VEC employs an attention mechanism to assign appropriate weights to the AST paths. **CODE2SEQ** [8] also utilizes AST paths to represent code snippets. It employs an encoder-decoder model along with bi-LSTM units to embed the nodes of the AST paths and subsequently make sequential predictions. **CODEBERT** [11] and **GRAPHCODEBERT** [26] are pre-trained neural code models based on RoBERTa. These models are capable of understanding and representing both natural language and programming language. During their pre-training, a large dataset comprising 2.3 million functions with paired documentation from six programming languages was utilized. GRAPHCODEBERT further incorporates data flow information during the pre-training stage, which enhances the syntactic-level understanding of code. **PLBART** [37] is a transformer-based neural model that follows a similar encoder-decoder architecture as BART [40], consisting of 6 encoder layers and 6 decoder layers. It was pre-trained on an extensive dataset of 680 million code examples using three pre-training tasks: Token Masking, Token Deletion, and Token Infilling. **CODET5** [25] is also based on the encoder-decoder architecture but built upon T5 [41]. In comparison to PLBART, CODET5 increases the number of decoder layers to 12. Additionally, CODET5 introduces code-oriented pre-training tasks to enhance its ability to extract code structure information. LLMs, such as **CHATGPT** [38] and **CHATGLM2** [39], have drawn significant public attention due to their impressive performance on general-purpose tasks. Unlike the pre-trained models specifically designed for SE tasks, these general-purpose models contain billions of parameters and have been trained on enormous corpora, e.g. the 17 TB dataset used for GPT-3.5.

Metrics. To evaluate the effectiveness of the attacks, we conducted a thorough analysis of the impact of adversarial examples on the prediction outputs of the target models. We assessed the models' performance using evaluation metrics including accuracy and F1 score for classification tasks. For the code summarization task (①), we employed the BLEU-4 metric to evaluate the quality of the generated summaries. By comparing the performance of the models in both the original example set $\mathcal{O}$ and the transformed example set $\mathcal{T}$, we were able to observe varying degrees of impact on their prediction capabilities. These differences (Δ) were reflected in the evaluation metrics, highlighting the effectiveness of the attacks in influencing the models' outputs.

To assess the impact of the attacks, we introduced a metric called the *Attack Success Rate* (*ASR*), which measures the rate at which predicted labels are changed after the transformation process. A higher *ASR* indicates a more effective attack, as it demonstrates a greater number of altered predictions. In our definition, $\mathcal{D}_\mathcal{T}$ represents the collection of all transformations, P represents the predictions made by the target models, and L represents the true labels. Each code transformation can be represented as a pair of corresponding examples (x_i, x'_{ij}), where x_i belongs to the original example set $\mathcal{O}$, and x'_{ij} belongs to the transformed example set $\mathcal{T}$. Considering the entire dataset, we define the *ASR* as the ratio of transformed examples for which the prediction differs from the original prediction:

$$ASR = \frac{\sum_{(x_i, x'_{ij}) \in \mathcal{D}_\mathcal{T}} \mathbb{I}(P(x_i) \neq P(x'_{ij}))}{|\mathcal{D}_\mathcal{T}|}$$

where $\mathbb{I}(P(x_i) \neq P(x'_{ij}))$ is an indicator function that evaluates to 1 if the predictions for x_i and x'_{ij} differ, and 0 otherwise. This definition allows us to quantify the success rate of the attacks in terms of how many predictions were altered compared to the original predictions.

In addition to the Attack Success Rate (*ASR*), we also incorporated the concept of *Robustness* as a measurement of the models' resilience to adversarial examples. We defined *Robustness* as the ability of a model to maintain its correct prediction when exposed to adversarial examples. If a model changed its correct prediction over an adversarial example, it was considered an indication of a lack of robustness. Formally, we define *Robustness (Ro)* as follows, where $\mathbb{I}(P(x_i) = L(x_i) \wedge P(x'_{ij}) \neq L(x_i))$ is another indicator function that evaluates if the model changes its prediction on correct examples:

$$Ro = \frac{\sum_{(x_i, x'_{ij}) \in \mathcal{D}_\mathcal{T}} \mathbb{I}(P(x_i) = L(x_i) \wedge P(x'_{ij}) \neq L(x_i))}{|\mathcal{D}_\mathcal{T}|}$$

Baselines. To better understand the effectiveness of *ALANCA*, we select BERT-Attack [42] and the previous SoTA code model attack algorithm CodeAttack [15] as baseline attack models for comparison. BERT-Attack is an NLP-based attack method that replaces the names of local variables and function parameters based on word importance ranks assigned by BERT. CodeAttack, on the other hand, is a recently introduced SoTA attack algorithm on code models that substitutes code tokens with specific constraints.

Implementation. In our evaluation, we fine-tuned all target models for each specific downstream task, while keeping the hyperparameters consistent with their original implementation. For CODE2VEC and CODE2SEQ, we performed fine-tuning for 50 epochs using a learning rate of 0.001 and a batch size of 512. Regarding the transformer-based models (CODEBERT, GRAPHCODEBERT, PLBART, and CODET5), we fine-tuned them on code-to-code tasks by utilizing specific embedded features. For CODEBERT and GRAPHCODEBERT, we used the embedded feature from the [CLS] vector to perform

TABLE III
EVALUATION RESULTS OF THE *ALANCA* ATTACK FRAMEWORK AGAINST EIGHT MODELS AND FOUR TASKS

Task	Metric	AST-based		Transformer-based				LLM	
		CODE2VEC	CODE2SEQ	CODEBERT	GCBERT	PLBART	CODET5	GPT-3.5	CHATGLM 2
Code Summarization	BLEU	Task Not Support		17.52	17.98	18.13	18.85	10.13	8.67
	Δ_{BLEU}			15.89↓	16.52↓	16.63↓	14.66↓	4.90↓	4.75↓
	ASR			85.8%	87.5%	88.3%	78.3%	53.5%	52.9%
	Ro			8.2%	8.0%	8.2%	20.6%	48.2%	46.5%
Method Name Prediction	F1	74.0	74.3	50.1	54.0	48.4	56.3	37.2	31.2
	Δ_{F1}	73.8↓	73.7↓	41.2↓	42.8↓	39.6↓	42.7↓	24.5↓	22.0↓
	ASR	99.9%	99.3%	81.2%	78.3%	80.8%	74.9%	63.6%	70.1%
	Ro	0.1%	0.2%	17.6%	20.4%	18.4%	14.0%	33.5%	27.3%
Code Classification	Acc	78.5	81.1	95.6	95.2	91.8	94.9	Task Not Support	
	Δ_{Acc}	63.2↓	61.9↓	72.7↓	71.4↓	64.4↓	55.0↓		
	ASR	77.2%	75.9%	74.3%	71.6%	66.8%	53.5%		
	Ro	19.5%	22.6%	24.0%	25.0%	29.9%	43.1%		
Clone Detection	F1	72.3	78.5	91.4	93.6	98.2	93.8	50.8	37.7
	Δ_{F1}	42.6↓	51.0↓	85.3↓	84.9↓	62.7↓	75.8↓	28.2↓	11.5↓
	ASR	53.1%	60.0%	92.6%	88.5%	69.5%	80.3%	53.9%	42.4%
	Ro	48.9%	37.4%	5.6%	9.1%	36.2%	18.8%	49.2%	53.7%

prediction. For PLBART and CODET5, we utilized the output of the last encoder layer to perform the prediction. For code-to-text tasks like Code Summarization (①), we added an additional decoder to the CODEBERT and GRAPHCODEBERT models to enable regressive prediction. The fine-tuning process for both transformer-based models lasted approximately 4-7 days each. To optimize the fine-tuning of CODEBERT and GRAPHCODEBERT, we used a learning rate of 0.0001 and a batch size of 32. Early stopping and learning rate decay (exponential) were implemented to prevent over-optimizing during training. Regarding the fine-tuning of PLBART and CODET5, we relied on the original processing and fine-tuning scripts provided by their respective implementations. In addition to evaluating the overall robustness of Large Language Models (LLMs) in code comprehension tasks, we also conducted a specific evaluation of their zero-shot performance. This evaluation was carried out in two LLMs, namely GPT-3.5 and CHATGLM2, in three code comprehension tasks: Code Summarization (①), Method Name Prediction (②), and Code Clone Detection (④). For each task, we carefully designed prompts to guide the LLMs' responses and implemented the measurement and adversarial attack processes using their Python APIs. This allowed us to assess the models' zero-shot performance and understand their vulnerabilities in different code comprehension scenarios.

Our fine-tuning and subsequent attack experiments were conducted on a server with a 128-core vCPU, 128 GiB RAM, and 4 NVIDIA A10 GPUs, running on Ubuntu 20.04 LTS. We have open-sourced this research at the anonymous repository https://anonymous.4open.science/r/NCMAA_release.

B. RQ 1 (Attack Effectiveness)

The performance of eight target models in four downstream tasks before and after adversarial attacks by *ALANCA* is presented in Table III. The table highlights the effectiveness of *ALANCA* in generating adversarial examples and misleading the target models. On average across all models, *ALANCA* achieved average Attack Success Rates (ASRs) of 74.3%, 81.0%, 69.8%, and 60.9% on the four code comprehension tasks. Notably, *ALANCA* successfully misled the CODE2VEC, CODE2SEQ, CODEBERT, and GRAPHCODEBERT models in more than 95% of their correct predictions for both the Method Name Prediction and Clone Detection tasks. The effectiveness of the attack performed by *ALANCA* varied across different tasks and models. It was observed that *ALANCA* achieved better effectiveness in simpler tasks like Method Name Prediction and on transformer-based code models. This can be attributed to the similarity in structure between these models and the discriminator and selector components in *ALANCA*, enabling better exploitation of vulnerabilities. In the case of attacks against Large Language Models (LLMs), the average Attack Success Rate was 55.6%, which was lower compared to attacks against other types of models. This can be attributed to the relatively lower effectiveness of LLMs on the supported code comprehension tasks and the much larger model sizes of LLMs compared to the models used in *ALANCA*.

In comparison, *ALANCA* demonstrates superior performance compared to the baseline attack methods, as illustrated in Table IV, in the majority of cases (11 out of 14). It is important to note that the adversarial examples generated by the baseline methods, namely BERT-Attack and CodeAttack,

TABLE IV
COMPARATIVE RESULT OF DIFFERENT ATTACK METHODS

Target Model	Attack Method	① Summarization			② MNP			③ Classification			④ Clone Detection		
		BLEU	***ASR***	***Ro***	**F1**	***ASR***	***Ro***	**Acc**	***ASR***	***Ro***	**F1**	***ASR***	***Ro***
CODE2SEQ	/	Task Not Support			74.3	/	/	85.1	/	/	78.5	/	/
	BERT-Attack				10.7	84.7%	14.5%	33.5	56.4%	42.7%	40.6	41.3%	53.1%
	RS				7.8	89.3%	8.4%	24.0	71.3%	30.6%	32.7	54.5%	44.2%
	w/o M_{ts}				2.7	93.1%	7.9%	22.6	70.9%	28.8%	28.9	63.4%	35.7%
	w/o M_d				4.5	93.5%	10.3%	19.5	75.0%	24.8%	31.1	57.9%	40.2%
	ALANCA				**0.4**	**99.3%**	**0.2%**	**19.2**	**75.9%**	**22.6%**	**27.5**	**60.0%**	**37.4%**
CODEBERT	/	17.52	/	/	50.1	/	/	95.6	/	/	91.4	/	/
	BERT-Attack	5.91	64.5%	39.7%	14.1	69.6%	32.2%	44.0	49.1%	47.3%	53.8	47.6%	58.0%
	CodeAttack	2.67	82.4%	15.8%	11.1	76.1%	22.4%	32.9	67.3%	34.9%	Not Evaluated		
	RS	3.64	77.9%	20.5%	12.9	77.2%	25.0%	34.1	64.1%	35.7%	12.0	87.6%	14.1%
	w/o M_{ts}	2.01	**86.9%**	12.5%	**8.5**	**82.3%**	**16.9%**	25.5	73.1%	26.7%	11.8	87.9%	14.4%
	w/o M_d	2.53	82.8%	16.4%	14.2	71.9%	26.4%	23.7	76.0%	24.8%	8.4	89.9%	9.5%
	ALANCA	**1.63**	85.8%	**11.2%**	8.9	82.6%	17.6%	**22.9**	**74.3%**	**24.0%**	**6.1**	**92.6%**	**5.6%**
CODET5	/	18.85	/	/	56.3	/	/	94.9	/	/	93.8	/	/
	BERT-Attack	8.62	64.3%	40.7%	18.4	69.3%	34.9%	40.9	58.4%	42.9%	42.6	52.5%	46.0%
	CodeAttack	5.31	71.0%	27.9%	13.1	76.7%	22.1%	39.0	57.0%	41.1%	Not Evaluated		
	RS	6.78	61.8%	36.9%	**9.1**	**84.7%**	**15.0%**	39.9	53.5%	43.1%	30.3	74.1%	34.6%
	w/o M_{ts}	4.61	**80.2%**	23.4%	11.7	77.5%	20.1%	41.2	56.1%	43.4%	28.7	70.0%	30.2%
	w/o M_d	5.99	65.2%	33.4%	13.6	74.9%	23.8%	39.3	38.9%	41.4%	20.0	79.2%	21.2%
	ALANCA	**4.19**	78.3%	**20.6%**	10.5	82.0%	17.9%	**36.0**	**62.8%**	**37.9%**	**18.0**	**80.3%**	**18.8%**
GPT-3.5	/	10.13	/	/	37.2	/	/	Task Not Support			50.8	/	/
	BERT-Attack	**5.18**	**53.9%**	50.7%	18.2	49.1%	48.4%				30.7	36.8%	61.3%
	RS	7.93	25.1%	78.4%	14.6	64.3%	37.1%				22.3	64.7%	42.0%
	w/o M_{ts}	7.81	28.3%	76.0%	14.2	63.8%	39.4%				23.7	52.7%	46.4%
	w/o M_d	6.16	39.4%	60.8%	**10.7**	**73.8%**	**29.0%**				**20.2**	**56.7%**	**40.1%**
	ALANCA	5.23	53.5%	**48.2%**	12.7	63.6%	33.5%				22.6	53.9%	49.2%

often fail to preserve the functionality of the code snippets and may even introduce syntactic errors. On the other hand, *ALANCA* excels not only in attack effectiveness but also in the quality of the generated adversarial examples. This can be attributed to the strict constraint on the range of perturbations enforced by *ALANCA*.

> ***Answer to RQ 1 (Attack Effectiveness)***: The attack results demonstrate the effectiveness of *ALANCA* in generating adversarial examples and misleading target models in various code comprehension tasks. The success rates vary depending on the complexity of the task and the target model's characteristics.

C. RQ 2 (Attack Efficiency)

The attack process curve in Figure 2 illustrates the performance of *ALANCA* in generating adversarial examples against CODE2SEQ, CODEBERT, and GPT-3.5 models for the Method Name Prediction task. In each attack epoch or iteration, *ALANCA* generated adversarial examples using only 0.1% of the entire testing set. The curve demonstrates that *ALANCA* efficiently performs the attack process and produces high-quality adversarial examples.

For the three target models mentioned, *ALANCA* required 21, 35, and 46 iterations, respectively, to generally achieve the attack objective. This translates to a query time of approximately 42, 70, and 92 with respect to the target models. The efficient generation of adversarial examples by *ALANCA* is evident from these results. When compared to the baseline CodeAttack algorithm, *ALANCA* exhibits significantly improved efficiency in terms of

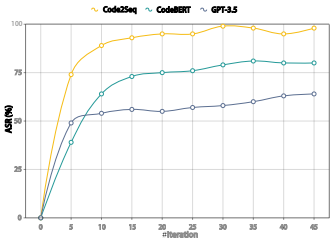

Fig. 2. Curves of attack efficiency and success rate in the ② MNP task

query time to the target models. On average, *ALANCA* achieves a query time to the CODEBERT and CODET5 models that are only 10% and 23%, respectively, of the query time required by the baseline CodeAttack algorithm.

> ***Answer to RQ 2 (Attack Efficiency):*** In general, the results demonstrate the efficiency of *ALANCA* in generating high-quality adversarial examples while minimizing the query time to the target models.

D. RQ 3 (Ablation Study)

Based on the results of the ablation experiments (in Table IV), it is observed that components of *ALANCA* have a significant impact on their effectiveness in generating adversarial examples. In the ablation study, the randomized model (RS) serves as the baseline model, and its adversarial ranking process and token selection are made randomly. Comparing the results of *ALANCA* with the RS baseline, it is shown that *ALANCA* outperforms the baseline in 52 out of 56 experiments, indicating the effectiveness of its components. The effectiveness of M_d (adversarial discriminator component) and M_{ts} (adversarial selector component) varies across different target models and downstream code comprehension tasks. For example, in attacks on Code Summarization and Method Name Prediction tasks against models like CODE2SEQ, CODEBERT, and CODET5, *ALANCA* without M_{ts} (adversarial selector) performs better than *ALANCA* without M_d (adversarial discriminator). This suggests that the adversarial discriminator component plays a major role in these cases. However, in simpler tasks, *ALANCA* without M_d (adversarial discriminator) shows better performance. This could be attributed to overfitting in these relatively larger target models, which may cause them to understand code in a more textual manner rather than considering the semantic and inner syntax of code. This situation is observed consistently in all downstream tasks of Large Language Models (LLMs), as these models perform these tasks in a zero-shot scenario. Due to the lack of task-specific training, LLMs may rely more on surface-level syntactic patterns rather than understanding the underlying semantics of the code.

> ***Answer to RQ 3 (Ablation Study):*** Both the adversarial discriminator component (M_d) and the token selector component (M_{ts}) demonstrate significant effectiveness in the attack process. Meanwhile, the effectiveness of these components varies across different target models and downstream code comprehension tasks.

V. THREATS TO VALIDITY

Threats to Internal Validity can arise from the implementation of *ALANCA* and the fine-tuning process involved in preparing the target models. To mitigate potential issues related to implementation, we have dedicated effort to building *ALANCA* using Spoon and PyTorch-lightning. Additionally, we have performed experiments to compare the performance of our target models used with the results reported in the original papers across all datasets and tasks involved. Threats to External Validity lie in the selected target neural models, downstream tasks, and relative datasets. In this research, we include four different code comprehension tasks and eight code models of three different types to make a thorough evaluation of our proposed method. Threats to Construct Validity relate to using evaluation metrics. In addition to the original metric used for the downstream tasks, we have included two additional metrics to ensure a comprehensive and clear presentation of the attack results.

VI. RELATED WORK

Several recent works proposed methods for adversarial attacks on code. [14], [16] studied the factors that affect the robustness of the code model. They proved that the predictions of neural models might change with text modifications (such as variables renaming and statements inserting), and the robustness of models may improve through appropriate adversarial training. [24] proposed a gradient-based adversarial attack method called DAMP, and demonstrated it's possible to perform targeted attacks in the discrete domain of code under white-box settings. [18]–[20] performed code transformations on AST several times to generate adversarial examples. [34], [43], [44] explored the task of malware detection models with small changes made on input. [45] used modified code repositories to poison training datasets and misled neural models to provide insecure code suggestions. As applications of adversarial examples, [12], [46] performed data augmentation using code transformations and designed contrastive learning processes using augmented code sets. [21] first proposed an iterative optimization-based technique for robustness detection and measurement on DL code models. [17] focused on attacking code comment generation models and proposed another iterative identifier substitution approach to generate adversarial examples. Like DAMP in [24], [22] proposed a white-box target attack to code models. [15] introduced a greedy search mechanism to substitute contextualized tokens in code snippets and performed black-box adversarial attacks.

VII. CONCLUSION

In this paper, we propose *ALANCA*, an active-learning guided adversarial attack framework for neural code models. By performing semantic-preserving translations on code and further refining with a designed adversarial discriminator and token selector, *ALANCA* can conduct adversarial attacks with high success rates, high generation quality, and good adaptability to different targets. Specifically, we carry out in-depth evaluations on four different code-understanding tasks to demonstrate the effectiveness of *ALANCA*. It's shown that *ALANCA* can successfully confuse and neutralize diverse neural models effectively and efficiently, including pre-trained models and LLMs used in software engineering.

REFERENCES

[1] M. Pradel and K. Sen, "Deepbugs: a learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *ICLR*. OpenReview.net, 2018.

[3] Y. Li, S. Wang, T. N. Nguyen, and S. V. Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 162:1–162:30, 2019.

[4] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *AAAI*. AAAI Press, 2016, pp. 1287–1293.

[5] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *ICSE*. IEEE / ACM, 2019, pp. 783–794.

[6] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M.-F. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 2091–2100.

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[8] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[9] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *NeurIPS*, 2019, pp. 6559–6569.

[10] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," in *ICLR (Poster)*. OpenReview.net, 2019.

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, 2020, pp. 1536–1547.

[12] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

[13] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[14] P. Bielik and M. Vechev, "Adversarial robustness for code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 896–907.

[15] A. Jha and C. K. Reddy, "Codeattack: Code-based adversarial attacks for pre-trained programming language models," in *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, B. Williams, Y. Chen, and J. Neville, Eds. AAAI Press, 2023, pp. 14 892–14 900. [Online]. Available: https://doi.org/10.1609/aaai.v37i12.26739

[16] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 1169–1176.

[17] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. C. Gall, "Adversarial robustness of deep code comment generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 60:1–60:30, 2022. [Online]. Available: https://doi.org/10.1145/3501256

[18] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," *arXiv preprint arXiv:2002.03043*, 2020.

[19] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," *arXiv preprint arXiv:2101.07910*, 2021.

[20] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of Neural Program Models with respect to semantic-preserving program transformations," *Information and Software Technology*, vol. 135, p. 106552, 2021.

[21] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, "Towards robustness of deep program processing models - detection, estimation, and enhancement," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, pp. 50:1–50:40, 2022. [Online]. Available: https://doi.org/10.1145/3511887

[22] F. Gao, Y. Wang, and K. Wang, "Discrete adversarial attack to models of code," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, pp. 172–195, 2023. [Online]. Available: https://doi.org/10.1145/3591227

[23] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U.-M. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," *arXiv preprint arXiv:2103.11882*, 2021.

[24] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[25] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.685

[26] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[27] J. Cao, M. Li, M. Wen, and S. Cheung, "A study on prompt design, advantages and limitations of chatgpt for deep learning program repair," *CoRR*, vol. abs/2304.08191, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.08191

[28] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *CoRR*, vol. abs/2304.07590, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.07590

[29] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2023, Melbourne, Australia, May 16, 2023*. IEEE, 2023, pp. 23–30. [Online]. Available: https://doi.org/10.1109/APR59189.2023.00012

[30] S. Bubeck, "Convex optimization: Algorithms and complexity," *Found. Trends Mach. Learn.*, vol. 8, no. 3-4, pp. 231–357, 2015. [Online]. Available: https://doi.org/10.1561/2200000050

[31] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 39–57. [Online]. Available: https://doi.org/10.1109/SP.2017.49

[32] W. Lin, Z. Yang, X. Chen, Q. Zhao, X. Li, Z. Liu, and J. He, "Robustness verification of classification deep neural networks via linear programming," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 11 418–11 427. [Online]. Available: http://openaccess.thecvf.com/content_CVPR_2019/html/Lin_Robustness_Verification_of_Classification_Deep_Neural_Networks_via_Linear_Programming_CVPR_2019_paper.html

[33] C. Casalnuovo, E. T. Barr, S. K. Dash, P. Devanbu, and E. Morgan, "A theory of dual channel constraints," in *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 25–28. [Online]. Available: https://doi.org/10.1145/3377816.3381720

[34] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based iot malware detection systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1296–1305.

[35] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436

[36] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. R. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html

[37] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tür, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Association for Computational Linguistics, 2021, pp. 2655–2668. [Online]. Available: https://doi.org/10.18653/v1/2021.naacl-main.211

[38] A. Bahrini, M. Khamoshifar, H. Abbasimehr, R. J. Riggs, M. Esmaeili, R. M. Majdabadkohne, and M. Pasehvar, "Chatgpt: Applications, opportunities, and threats," *CoRR*, vol. abs/2304.09103, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.09103

[39] A. Zeng, X. Liu, Z. Du, Z. Wang, H. Lai, M. Ding, Z. Yang, Y. Xu, W. Zheng, X. Xia *et al.*, "Glm-130b: An open bilingual pre-trained model," *arXiv preprint arXiv:2210.02414*, 2022.

[40] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, Eds. Association for Computational Linguistics, 2020, pp. 7871–7880. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.703

[41] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[42] L. Li, R. Ma, Q. Guo, X. Xue, and X. Qiu, "BERT-ATTACK: adversarial attack against BERT using BERT," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, 2020, pp. 6193–6202. [Online]. Available: https://doi.org/10.18653/v1/2020.emnlp-main.500

[43] H. Alasmary, A. Abusnaina, R. Jang, M. Abuhamad, A. Anwar, D. Nyang, and D. Mohaisen, "Soteria: Detecting adversarial examples in control flow graph-based malware classifiers," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 888–898.

[44] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye, "Interpreting deep learning-based vulnerability detector predictions based on heuristic searching," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–31, 2021.

[45] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," *CoRR*, vol. abs/2007.02220, 2020.

[46] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," *arXiv preprint arXiv:2009.02731*, 2020.

[47] S. Cheng, Y. Dong, T. Pang, H. Su, and J. Zhu, "Improving black-box adversarial attacks with a transfer-based prior," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10 932–10 942.

[48] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," *CoRR*, vol. abs/1606.04435, 2016.

[49] Z. Huang and T. Zhang, "Black-box adversarial attack with transferable model-based embedding," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[50] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," in *26th European Signal Processing Conference, EUSIPCO 2018, Roma, Italy, September 3-7, 2018*. IEEE, 2018, pp. 533–537.

[51] C. Mayer and R. Timofte, "Adversarial sampling for active learning," in *IEEE Winter Conference on Applications of Computer Vision, WACV 2020, Snowmass Village, CO, USA, March 1-5, 2020*. IEEE, 2020, pp. 3060–3068.

[52] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 2016, pp. 372–387.

[53] D. Yoo and I. S. Kweon, "Learning loss for active learning," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 93–102.

[54] W. E. Zhang, Q. Z. Sheng, A. Alhazmi, and C. Li, "Adversarial attacks on deep-learning models in natural language processing: A survey," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 11, no. 3, pp. 1–41, 2020.

[55] B. Zhang, L. Li, S. Yang, S. Wang, Z.-J. Zha, and Q. Huang, "State-relabeling adversarial active learning," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. IEEE, 2020, pp. 8753–8762.

[56] F. Wu, A. H. S. Jr., T. Zhang, C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," in *ICML*, ser. Proceedings of Machine Learning Research, vol. 97. PMLR, 2019, pp. 6861–6871.

[57] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR (Poster)*, 2015.

[58] A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru, and F. Roli, "Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks," in *USENIX Security Symposium*. USENIX Association, 2019, pp. 321–338.