Exploiting the Adversarial Example Vulnerability of Transfer Learning of Source Code

Yulong Yang, *Graduate Student Member, IEEE*, Haoran Fan, Chenhao Lin, *Member, IEEE*, Qian Li, *Member, IEEE*, Zhengyu Zhao, *Member, IEEE*, and Chao Shen, *Senior Member, IEEE*

***Abstract*— State-of-the-art source code classification models exhibit excellent task transferability, in which the source code encoders are first pre-trained on a source domain dataset in a self-supervised manner and then fine-tuned on a supervised downstream dataset. Recent studies reveal that source code models are vulnerable to adversarial examples, which are crafted by applying semantic-preserving transformations that can mislead the prediction of the victim model. While existing research has introduced practical black-box adversarial attacks, these are often designed for transfer-based or query-based scenarios, necessitating access to the victim domain dataset or the query feedback of the victim system. These attack resources are very challenging or expensive to obtain in real-world situations. This paper proposes the cross-domain attack threat model against the transfer learning of source code where the adversary has only access to an open-sourced pre-trained code encoder. To achieve such realistic attacks, this paper designs the Code Transfer learning Adversarial Example (CodeTAE) method. CodeTAE applies various semantic-preserving transformations and utilizes a genetic algorithm to generate powerful identifiers, thereby enhancing the transferability of the generated adversarial examples. Experimental results on three code classification tasks show that the CodeTAE attack can achieve 30% ∼ 80% attack success rates under the cross-domain cross-architecture setting. Besides, the generated CodeTAE adversarial examples can be used in adversarial fine-tuning to enhance both the clean accuracy and the robustness of the code model. Our code is available at https://github.com/yyl-github-1896/CodeTAE/.**

***Index Terms*— Transfer learning, source code models, cross-domain adversarial attack, adversarial transferability.**

I. Introduction

PRE-TRAINED Deep Neural Network (DNN) models of source code are increasingly adopted in various soft engineering tasks, such as vulnerability detection, authorship attribution, code generation, etc [1], [2], [3]. State-of-the-art (SOTA) source code models [4], [5], [6], [7] commonly follow the pre-training & fine-tuning paradigm. Code feature encoders are first pre-trained on a large-scale dataset of source code with a self-supervised learning manner and then fine-tuned on a downstream task in a supervised manner.

Despite the success of the DNN models of source code, they are vulnerable to adversarial examples [8], which are crafted by adding human-imperceptible perturbations but can mislead the prediction of the victim model. The existence of adversarial examples may bring threats to the security of deep learning-based source code processing applications in the real world. For one instance, the adversary can use adversarial attacks to bypass the automated code copyright authentication tools based on deep learning [9]. For another instance, the adversary can submit a malicious program with vulnerabilities to a shared engineering project and bypass the detection of the intelligent code vulnerability detection tools [10] by crafting adversarial examples. Previous research has developed the adversarial attack by inserting program obfuscations into the code. However, there is no previous study, to our best knowledge, that focuses on the security risks of code models under the transfer learning setting, in which the adversary has only access to the source domain code encoder but has no knowledge about the victim domain used to train the downstream classifier.

Existing researchers only studied attacking the victim source code classifier within the same domain, which can be divided into transfer-based attacks [11], [12] and query-based attacks [13], [14], [15], [16], [17], [18], [19]. The transfer-based attacks assume the adversary has full access to the training data of the victim model. The adversary can train a local surrogate model to generate adversarial examples and leverage the adversarial transferability to attack the victim model [20], [21], [22]. The query-based attacks assume the adversary has the query permission of the victim system. The adversary can design heuristic searching algorithms to generate adversarial examples under the guide of the output score of the victim system. Besides, many previous query-based attacks also rely on pre-trained source code encoders to generate semantic-preserving transformations [15], [16], [17]. However, these black-box attacks are still not applicable to attacking realistic pre-training & fine-tuning systems because the downstream training dataset or the output score feedback is hard (or very expensive) to obtain.

Manuscript received 24 December 2023; revised 13 April 2024; accepted 7 May 2024. Date of publication 16 May 2024; date of current version 31 May 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB3100700; in part by the National Natural Science Foundation of China under Grant 62132011, Grant 62161160337, Grant 62206217, Grant 62376210, Grant U20A20177, and Grant U21B2018; and in part by Shaanxi Province Key Industry Innovation Program under Grant 2021ZDLGY01-02 and Grant 2023-ZDLGY-38. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Haijun Zhang. *(Yulong Yang and Haoran Fan contributed equally to this work.) (Corresponding author: Chenhao Lin.)*

Yulong Yang, Chenhao Lin, Qian Li, Zhengyu Zhao, and Chao Shen are with the School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: linchenhao@xjtu.edu.cn).

Haoran Fan is with the School of Software Engineering, Xi'an Jiaotong University, Xi'an 710049, China.

Digital Object Identifier 10.1109/TIFS.2024.3402153

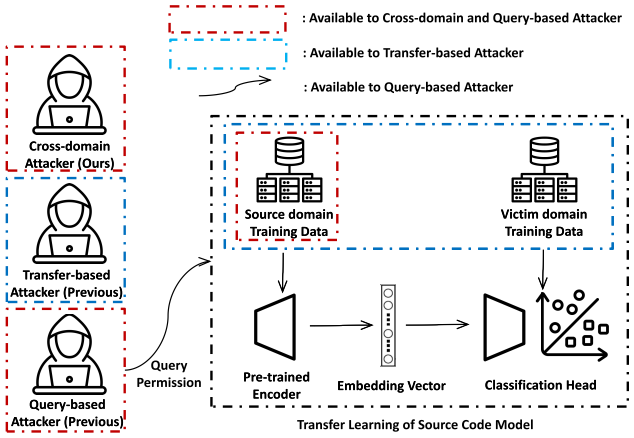


Fig. 1. Illustration and comparison between the query-based black-box attack, the transfer-based black-box attack, and our cross-domain attack.

In vision systems, researchers have developed cross-domain transfer-based attacks that utilize only pre-trained models to attack downstream classifiers trained on unknown domains [23], [24], [25], [26], this attack setting is practical because of the widely available pre-trained models in the open-source community. Instead of sampling a large-scale training dataset to train a surrogate model, the cross-domain attack only assumes the adversary has a limited number of test examples for crafting adversarial examples, which is assumed to be more realistic [15], [17], [27]. However, to our best knowledge, there is no previous literature studying the cross-domain transfer-based attacks in the source code tasks.

To address this research gap, this paper defines the cross-domain attack threat model against transfer learning of source code. The cross-domain attack only assumes the adversary has access to an open-sourced pre-trained code encoder to generate adversarial examples with limited test examples, illustrated in Fig. 1. Compared to previous transfer-based and query-based attacks, the cross-domain threat model is more challenging because the adversary can no longer obtain guiding information from either querying or the surrogate model. To this end, this paper proposes the code Transfer learning Adversarial Example (CodeTAE) method to generate adversarial examples under the cross-domain setting. Specifically, CodeTAE guides the optimization of the adversarial example by maximizing the feature divergence between the adversarial and clean examples in the intermediate feature space of the surrogate encoder. To enhance the attack transferability, CodeTAE designs a genetic algorithm-based substitute identifier generation method, powering the various program obfuscation operations used in generating effective adversarial examples.

We evaluate the attack performance of CodeTAE on multiple code classification tasks. The experimental results demonstrate that CodeTAE is capable of producing highly transferable adversarial examples against unknown victim models while maintaining the stealthiness of the adversarial examples. Finally, we use the generated CodeTAE adversarial examples to adversarially fine-tune the code model. The fine-tuning results show that the generated CodeTAE adversarial examples can enhance both the robustness and the clean accuracy of the model. In sum, our contributions are as follows.

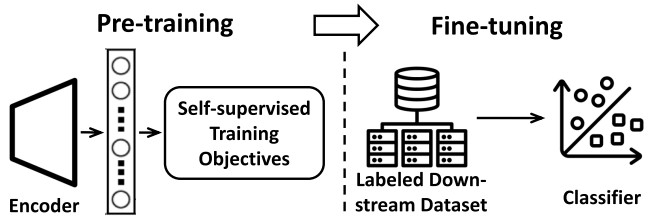


Fig. 2. Overview of the pre-training & fine-tuning paradigm of code models.

- We are the first to explore the vulnerability of the transfer learning of source code models in a more practical cross-domain setting.
- We design the CodeTAE attack to show the feasibility of generating cross-domain adversarial examples by utilizing the genetic algorithm to enhance the attack transferability.
- Extensive experimental results verify the attack effectiveness of the proposed CodeTAE in attacking unknown victim models. Besides, CodeTAE is beneficial in enhancing both the robustness and accuracy of the source code model through adversarial fine-tuning.

II. Background and Related Work

A. Transfer Learning of Source Code

The transfer learning paradigm is widely adopted in SOTA DNN models of source code, in which the code models are first pre-trained on the source-domain dataset and then fine-tuned on the downstream datasets, such as the authorship attribution [1], the clone detection [2], and the vulnerability detection [28]. At the pre-training stage, a code encoder is trained on a multi-language source code corpus with self-supervised training techniques. The pre-trained code encoder converts the input code example x into a high-dimensional feature embedding vector $h \in R^n$. At the fine-tuning stage, a classifier is attached after the code encoder to output a prediction based on the extracted feature vector x. The whole model (encoder + classifier) is fine-tuned on the downstream dataset in a supervised manner. The whole process of the pre-training, fine-tuning paradigm is illustrated in Fig. 2.

Previous works have put much effort into developing powerful pre-trained code encoders. The Code2Vec [29] is a pioneering work in this field, which extracts Abstract Syntax Tree (AST) from source code to generate the feature vector. Code2Seq [30] is a Seq2Seq variant of Code2Vec that can be utilized in generation tasks. Later on, inspired by the progress in Natural Language Processing [3], several works tried to build programming language representation directly from raw code tokens. For instance, CuBERT [31] leveraged the powerful masked language modeling capability of BERT to learn generic programming language representation. Later on, CodeBERT [4] further added a replaced token detection task to learn NL-PL (natural language and programming language) representation. GraphCodeBERT [5] improves the CodeBERT by further adding the structural representation based on the

data flow. Besides the BERT-like architecture, GPT [32] and UniLM [33] are employed to enhance the model performance of code completion tasks. Different from the above models, CodeT5 [6] and CodeT5+ [7] are encoder-decoder architectures that optimize the computational structure of the encoder and achieve SOTA performance in various downstream tasks. In the experiment section of this paper, we select three representative model architectures CodeBERT, GraphCodeBERT, and CodeT5 as the victim models of the adversary. But please note that the proposed attack is general and can be used to attack any other victim models.

B. Query-Based Adversarial Attacks Against Code Models

Despite the success of the above pre-trained code models, they are found to be vulnerable to adversarial examples [20], which are crafted by adding human-imperceptible perturbations and can mislead the prediction of the victim model. The realistic adversarial attacks discussed in the previous studies can be divided into query-based adversarial attacks and transfer-based adversarial attacks. In the realistic setting, the adversary usually has no access to the parameters and gradients of the victim model. One of the major techniques of achieving a realistic attack is the query-based attack, which assumes the adversary can query the victim model and approximate the gradient direction using the query feedback. The query-based adversary can also adopt the open-sourced pre-trained code encoder to generate code transformations. In terms of the feedback information, the query-based attacks can be further divided into the score-based attacks [13], [14], [15], [16], [17] and decision-based attacks [34]. The former assumes that the victim system outputs the confidence score of the current input, while the latter assumes that only the prediction label is provided. Zhang et al. [13] applies identifier renaming as the code transformation operation, and adopts a Metropolis-Hastings sampling-based Method (MHM) to search the optimal adversarial examples. The CodeBERT attack [14] uses pre-trained CodeBERT to generate new identifiers to guarantee the naturalness and stealthiness of the adversarial code and use the greedy search to optimize. ALERT [15] further combines the genetic algorithm into the substitute identifier searching algorithm to enhance the attack success rates and reduce the number of queries. An explanation-based method is adopted to find the optimal attack position in CodeBERT attack [14] and ALERT [15]. While all the above studies only adopt identifier renaming as the semantic-preserving transformation, Tian et al. [17] further consider structural code transformation operations and leveraging code references to substantially enhance the attack success rates of the targeted attack. Li et al. [34] develop a decision-based query attack against a Functional Call Graph (FCG) based Android malware detection neural model, which is a more challenging setting for the adversary than the score-based attacks. Please note that previous query-based attacks also assume the white-box accessibility of the pre-trained code encoder to ensure the naturalness of the attack. Thus, the adversary's capability in this paper is a subset of that of the query-based attacks.

C. Transfer-Based Adversarial Attacks Against Code Models

The transfer-based adversarial attack does not need querying the victim model but requires training a local surrogate model with the same training dataset as the victim model. The adversary can generate adversarial examples by attacking the surrogate model and leverages the adversarial transferability [20] (i.e., adversarial examples successful on one model are probably successful on another model) to attack the victim model. The transferability of adversarial examples against code models is first discussed in [11], where the adversarial examples are generated on the surrogate model with the Monte-Carlo Tree search and are transferable across different victim models. Liu et al. [12] propose a transfer-based attack SCAD against the source code authorship attribution task by applying various code transformation operations, including the arithmetically equivalent, the dead code injection, etc.

Different from the previous query-based attacks and transfer-based attacks, this paper proposes a Code Transfer learning Adversarial Example (CodeTAE). Compared to traditional transfer-based attacks, cross-domain transfer-based attacks are more challenging because of the following aspects: (1) The cross-domain adversary cannot access the training dataset distribution of the victim model. Instead, the cross-domain adversary can only utilize a pre-trained surrogate model in a different domain. In comparison, the traditional transfer-based adversary can i.i.d. sample data from the victim model's training dataset; (2) The cross-domain adversary does not know the label distribution of the victim tasks, prohibiting the adversary from training an end-to-end surrogate classifier. Thus, we design an intermediate-level attack algorithm to craft transferable adversarial examples. We compare different transfer-based attack threat models in Tab. I.

D. Adversarial Defense for Code Models

Recent works have developed defense methods against adversarial attacks for neural models of source code from various perspectives, including adversarial training, adversarial fine-tuning [15], [35], input-prepossessing, and model ensembling. Adversarial training [35], [36], [37], [38], [39], [40] iteratively generates extreme white-box adversarial examples and trains a robust CLM with the min-max optimization formulation. Adversarial training is regarded as the strongest defense but may require expensive computational costs and harm the generalization capability of the model [41]. Adversarial fine-tuning augments the downstream dataset by adding semantic-preserving transformations. The difference between adversarial fine-tuning and adversarial training is that the augmented examples are not necessarily extreme white-box adversarial examples and, thus may improve both the robustness and generalization of the model. The input-denoising defenses [35], [39], [42], [43] are applied at the inference stage of CLM to filter out adversarial perturbations in the input code snippet and provide the correct outputs. Model ensembling [44] was proposed to be combined with other defense methods to further improve the robustness of CLMs. This paper explores the effectiveness of using the generated CodeTAE adversarial examples in the adversarial fine-tuning

TABLE I
COMPARISON OF DIFFERENT THREAT MODELS FOR BLACK-BOX ATTACKS AGAINST DNN MODELS OF SOURCE CODE

Threat Model	Representative Work	Adversary's Accessibility		
		Open-sourced Source Domain Data	Private Victim Domain Data	Querying the Victim Model
Transfer-based Attack	[11], [12]	✓	✓	×
Query-based Attack	[13]–[16], [34]	✓	×	✓
Cross-domain Attack	Ours	✓	×	×

process and finds that both the robustness and generalization of the model can be enhanced.

E. Adversarial Attacks Against Vision Transfer Learning

Adversarial Attacks against the transfer learning paradigm are first studied in vision systems. Wang et al. [45] discovered that utilizing pre-trained vision models to fine-tune the downstream dataset increases the adversarial risks no matter which fine-tuning strategy is applied. Specifically, they assumed that the adversary has white-box access to the open-sourced pre-trained models, which are also adopted in the victim model architecture. They leveraged the intermediate feature of the pre-trained surrogate model to guide the generation of adversarial examples. Luo et al. [46] enhanced the cross-domain transferability of adversarial examples from the perspective of frequency-domain transformation. Zhang et al. [26] utilized generative models to achieve cross-domain adversarial attacks against unknown image models. Sun et al. [47], [48] trained surrogate models with only several available test images to achieve transfer-based attacks against unknown victim models. The threat model most similar to ours is the work of Zhang et al. [25] and Ban and Dong [49], which further relaxed the assumption on the adversary's knowledge, assuming that the adversary only knows that the victim model is fine-tuned from an unknown pre-trained vision model. They also studied the cross-domain & cross-architecture transferability of the generated adversarial images. Despite the above progress in vision systems, to our best knowledge, there is no previous study on the adversarial risks of transfer learning of language models on the source code, which is the focus of this paper. Please note that attacking transfer learning of source code is a more challenging task than attacking the vision system because of the discrete nature and the strict syntax restriction of the source code.

III. THREAT MODEL AND OVERVIEW OF CODETAE

A. Threat Model: Cross-Domain Transfer-Based Attack

The cross-domain threat model in this paper is defined as follows in terms of the attack objective, the attack constraint, the adversary's knowledge, and the adversary's capability.

1) Attack Objective: The adversary aims to mislead the prediction of the victim models. This paper considers untargeted attack, i.e., making the victim model output the wrong label for the current attack input, which can be formalized as,

$$C(x_{adv}) \neq C(x) = y, \tag{1}$$

where $C(\cdot)$ denotes the victim classifier, x is the input example, x_{adv} is the adversarial example, and y is the true label.

2) Attack Constraint: To generate adversarial examples in the real world, the adversary is required to satisfy the semantic equivalence constraint. In other words, the adversary should make sure that the adversarial source code can be compiled correctly and does not change the original functionality of the source code, which can be formalized as,

$$S(x_{adv}) = S(x), \tag{2}$$

where S is the semantic equivalence checking function.

3) Adversary's Knowledge: The cross-domain setting assumes that the victim task is agnostic to the adversary, that is, the adversary has no access to the large-scale data of the downstream dataset. Besides, the adversary has only black-box knowledge about the victim model. In other words, the adversary does not know the architecture, training data, hyper-parameters, and other details about the victim model.

4) Adversary's Capability: The cross-domain threat model assumes the adversary can only collect pre-trained code encoders (e.g., Code2Vec [29], CodeBERT [4], etc.) but does not have the capability of training a surrogate classifier with the same training dataset of the victim model. The pre-trained code encoder F maps the input example x into a feature vector h in the n-dimensional Euclidean space, which is $F(x) = h \in \mathcal{R}^n$. The adversary can only generate adversarial examples with the guide of h but cannot train a surrogate model with the same training dataset as the victim model. The adversary has no permission to query the victim model.

Please note that our cross-domain threat model is different from the previous "query-based" or the "transfer-based" adversarial attack threat model, which is illustrated in Fig. 1 and Tab. I. The query-based attack allows the adversary to frequently query the victim model and get the output confidence score to guide the optimization of x_{adv}. The transfer-based attack assumes the adversary has the capability of training a surrogate classifier with the same training dataset as the victim model. The adversary in the transfer-based setting can generate adversarial examples on the local surrogate model and leverage the transferability [12], [20] of adversarial examples to attack the victim model. Both query-based attacks and transfer-based attacks can leverage pre-trained source code encoders in their attack generation process. We can see that the cross-domain threat model narrows the adversary's knowledge and capability, which brings the following new challenges compared to previous attacks.

Challenge 1: The adversary does not have a complete surrogate classifier, making the previous transfer-based attack optimization framework based on the cross-entropy loss and the gradients useless in this setting.

TABLE II
THE SEMANTIC EQUIVALENT CODE TRANSFORMATION OPERATIONS USED IN CODETAE

ID	Transformation Name	Type	Description and Examples (in Python)
1	Identifier renaming	Identifier renaming	Renaming the identifier in the code snippet
2	Insert unused variable	Deadcode insertion	Insert an unused variable definition at any line, i.e. *unused_var = 'hello world!'*
3	Insert empty print	Deadcode insertion	Insert an empty print statement at any line, i.e. *print('',end='')*
4	Insert empty if	Deadcode insertion	Insert an empty if statement, i.e. *if False: unused_var = 'hello world!'*
5	Insert empty while	Deadcode insertion	Insert an empty while statement, i.e. *while False: unused_var = 'hello world!'*

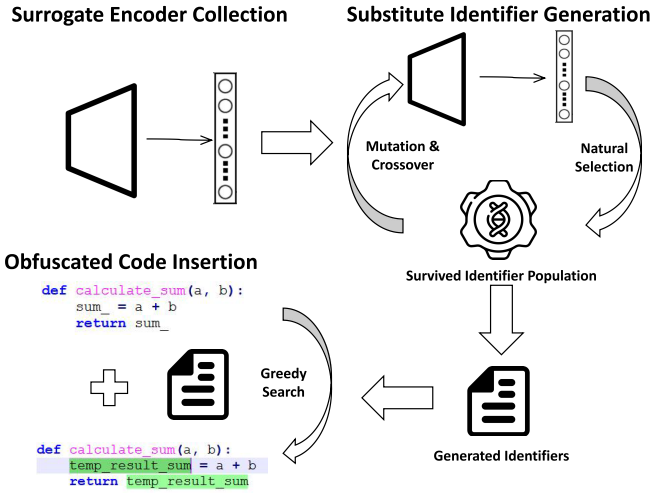


Fig. 3. Overview of the CodeTAE method, consisting of the surrogate encoder collection, the substitute identifier generation, and the obfuscated code insertion process.

Challenge 2: The adversary cannot query the victim model and use the query feedback to guide the attack process as in the previous query-based attacks.

Challenge 3: The strict constraints above require the adversary to seek more powerful program transformation operations while still preserving the original semantics and functionality of the source code.

B. Overview of CodeTAE

To address the above challenges, this paper proposes the Code Transfer learning Adversarial Example (CodeTAE) method, consisting of the surrogate encoder collection, the substitute identifier generation, and the obfuscated code insertion process, as illustrated in Fig. 3. Specifically, to address the challenge 1 & 2, the surrogate encoder collection process of CodeTAE sends the input code x_{adv} to the surrogate encoder F and gets the latent feature embedding $h = F(x_{adv})$, which can be used to guide the generation process of adversarial examples. To address challenge 3, the obfuscated code insertion process of CodeTAE prepares multiple semantic-preserving code transformation operations to boost the attack effectiveness, including identifier renaming, and dead code insertion. The substitute identifier generation process of CodeTAE further boosts the transferability of the generated adversarial code to guarantee the attack's effectiveness.

IV. METHODOLOGY

A. Surrogate Code Encoder Collection

CodeTAE collects pre-trained encoder F to convert the input code x_{adv} into the latent feature vector h, that is, $h = F(x_{adv})$. The adversary can then compare the adversarial code and the original code in the latent feature space to guide the generation process of adversarial examples, that is,

$$\mathcal{L}(x_{adv}) = ||F(x_{adv}) - F(x)||_2^2, \tag{3}$$

where $||\cdot||_2^2$ denote the squared l_2-norm, which calibrates the distance between x and x_{adv} in the latent feature space of F. Then, the problem of generating adversarial code in the cross-domain threat model can be formalized as,

$$\max_{x_{adv}} \quad \mathcal{L}, \quad s.t. \quad S(x_{adv}) = S(x), \tag{4}$$

where $S(x)$ denotes the semantics of the code snippet x.

Intuitively, if the adversary can achieve a high $\mathcal{L}$ value, then the generated adversarial code will probability have high attack success rates on the black-box victim model. This is because different code models share similar features in the shallow layers [50], [51]. Adversarial examples with high loss value on the surrogate encoder will probably also disrupt the shallow features of the victim model, despite their different model parameters and architectures [20]. The disruption in the shallow layer of the victim model will propagate till the last layer of the neural classifier [12], [20], which finally leads to the wrong prediction.

B. Code Transformation Operations

CodeTAE utilizes various code transformation operations to generate effective adversarial code while maintaining the semantic equivalence constraint of Eqn. 2. The code transformation operations used in CodeTAE are summarized in Tab. II. CodeTAE adopts five code transformation operations, including one identifier renaming and four deadcode insertions. The renamed identifier and the string variables used in the deadcode insertion can be optimized by the substitute identifier generation process of CodeTAE to further enhance the attack's effectiveness.

C. Substitute Identifier Generation

As mentioned above, the identifier renaming, unused variable insertion, empty if insertion, and the empty while insertion operation used in the CodeTAE have string variables that can be searched. The substitute identifier generation process searches powerful string variables that can be used in the attack code transformation using the Genetic Algorithm (GA). We use the vanilla variation of GA. Discussions on CodeTAE using other GA variations can be found in *Appendix*. The chromosome representation, seed preparation, fitness function, and mutation of our designed genetic algorithm are as follows.

1) Chromosome Representation: We set one string variable s as a chromosome, the generated s, in the end, can be used as the replaced identifier and to fill the string variable placeholder in the deadcode insertion. We set the maximum length of s as 20. Each chromosome s should be a legal variable name.

2) Seed Preparation: The initial population (a.k.a. seed) plays an important role in the performance of the genetic algorithm [52]. This paper collects seed variable names from the test code examples. Since we only collect 200 variable names from the test set, the requirement that the adversary has no access to the large-scale training dataset is still satisfied.

3) Fitness Function: We use the pre-trained code embedding model to calculate the fitness of each chromosome, which is defined as,

$$Fitness(s') = ||s - s'||_2^2, \tag{5}$$

where s is the original identifier extracted from the test code, and s' is the generated new identifier. s' with higher fitness value will be more likely to survive till the next population.

4) Mutation & Crossover: In each iteration, we generate a new chromosome with four mutation operations and one crossover operation, which is defined as follows.

Mutation: 1) Insert: insert a character from the alphabet at a random position in the chromosome; 2) Delete: delete a character at a random position in the chromosome; 3) Flip: randomly change a character in the chromosome into another one in the alphabet; 4) Swap: swap the characters from two random positions in the chromosome.

Crossover: The crossover operation randomly cuts two chromosomes to generate two new chromosomes. For instance, if the parent chromosomes are "aaabbb" and "cccddd", then the new child chromosomes can be "aaaddd", and "cccbbb". The cutting position is randomly selected.

The whole process of the genetic algorithm is summarized as Alg. 1. At each GA iteration, we randomly decide whether to use the crossover or the mutation operation to generate new children. If using the crossover, each pair of chromosomes in P will be used to generate two new children. If using the mutation, we randomly select one mutation operation for each chromosome in P to generate a new child. The generated children are added into P. The chromosomes in P will be then sorted in decreasing order in terms of the fitness value, and only the first M chromosomes in P can be preserved.

Algorithm 1 Substitute Identifier Generation Based on GA

Require: A pre-trained code encoder F, the number of GA iteration epochs N, the maximum size of the population M, crossover rate r, collected initial population P_0.

Initialize: Initialize P as P_0, calculate the fitness value for every chromosome in P_0.
Repeat for N iterations:
  $p = U(0, 1)$ # U denotes the uniform distribution
  if $p < r$:
    $child = crossover(P)$
  else:
    $child = mutation(P)$
  $P = P \cup child$
  calculate the fitness value of each item in P
  sort P in decreasing order of the fitness value
  preserve only the first M items in P
Return P

Algorithm 2 Obfuscated Code Insertion Based on GS

Require: A pre-trained code encoder F, the maximum number of insertable perturbation ϵ, a code snippet x, the generated identifier population P.

Collect all the identifier s in x as set S
Get the I of each identifier s in S with Eqn. 6
Sort the identifier s in S in the decreasing order of I
Initialize $x_{adv} = x$
Repeat for ϵ iterations:
  pop the top-1 item s in S
  $bestLoss = 0$ # temp variable used in search
  $bestx = x_{adv}$ # temp variable used in search
  For s' in P:
    Replace the s as s' in x and forms x_{adv}
    $loss = \mathcal{L}(x_{adv})$
    if $loss > bestLoss$:
      update $bestLoss$
      update $bestx$
  $x_{adv} = bestx$
Return x_{adv}

D. Obfuscated Code Insertion

The generated identifier population P can be used in the identifier renaming, unused variable insertion, and empty if/while insertion transformation operations. The empty print insertion operation does not need the identifier population P. At the obfuscated code insertion stage, we use an explanation-guided Greedy Search (GS) algorithm to insert the code transformation operations into the code snippet to generate adversarial examples. Please note that all the attack operations in this step are applied to the pre-trained code encoder only, without any access to the victim model on the downstream dataset. First, we select the optimal position in the code snippet for renaming/insertion by calculating the importance score of this position as follows,

$$I = ||F(x') - F(x)||_2^2, \tag{6}$$

where x' denotes the code snippet with a placeholder at the current position. The value of I reflects the importance of the current position on the behavior of the pre-trained surrogate encoder. Positions with higher I are more vulnerable and will be attacked first. Then, all the identifiers/placeholders in the code snippet will be sorted in the decreasing order of I. We apply the code transformation operations to the identifier/placeholder in terms of the order of the importance score until the maximum number of changes is reached. We use the greedy search to select the optimal candidate identifier in P to rename/insert the original code snippet. The whole process of the obfuscated code insertion (take the identifier renaming as an example) is summarized as Alg. 2.

E. Attack Success Verification

Remind that the cross-domain threat model defined in Sec. III assumes that the adversary only has access to the pre-trained code encoder F. The generated cross-domain adversarial examples should be verified on a victim model fine-tuned on an unknown downstream dataset. A successful attack should be able to subvert the correct prediction of the victim model into a wrong one. Specifically, we consider two occasions that may occur in the real-world cross-domain setting as follows.

1) Adversarial Transferability Within the Same Encoder: The pre-trained code encoders are widely adopted as the feature extractor of the classifier for the downstream code classification tasks. The pre-trained encoder used by the adversary and the victim model may have the same architecture. Please note that this case still belongs to the cross-domain threat model since the fine-tuned victim model parameters and its classification head are unknown to the adversary. The adversary generates the adversarial examples by maximizing the $\mathcal{L}$ on the surrogate encoder and tests the success rates of the generated adversarial examples on the victim model with both encoder and classifier.

2) Adversarial Transferability Across Different Encoders: This is the occasion where the encoder architecture adopted by the adversary is different from that of the victim model. This setting may be more challenging because the adversary has less information about the victim model. Under this occasion, the generated adversarial examples should have strong cross-domain transferability and cross-architecture transferability [12], [20] to achieve successful attacks.

TABLE III
STATISTICS OF EACH DATASET AND VICTIM MODEL

Dataset	Train/Val/Test	Model	Clean Acc. (%)
Authorship Attribution	528/–/132	CodeBERT	82.58
		GraphCodeBERT	79.55
		CodeT5	80.30
Clone Detection	90,102/4,000/4,000	CodeBERT	98.06
		GraphCodeBERT	97.04
		CodeT5	98.50
Defeat Detection	21,854/2,732/2,732	CodeBERT	60.25
		GraphCodeBERT	62.00
		CodeT5	64.25

TABLE IV
THE NUMBER OF CHANGES OF EACH CODE TRANSFORMATION OPERATION IN THE COMPLETE CODETAE ON EACH DATASET

Dataset	Transformation	number of changes
Authorship Attribution	Identifier renaming	2
	Insert unused variable	4
	Insert empty print	2
	Insert empty if	2
	Insert empty while	2
Clone Detection	Identifier renaming	2
	Insert unused variable	6
	Insert empty print	0
	Insert empty if	15
	Insert empty while	15
Defeat Detection	Identifier renaming	2
	Insert unused variable	2
	Insert empty print	2
	Insert empty if	2
	Insert empty while	4

V. EVALUATION

A. Experimental Setup

1) Datasets & Victim Models: The evaluation is conducted on three code classification datasets: Authorship Attribution (in Python) [1], Clone Detection (in Java) [2], and Defeat Detection (in C) [3]. We briefly summarize each dataset as follows: The authorship attribution task is to identify the authorship of a given code snippet. which originated from the Google Code Jam (GCJ) challenge. The clone detection aims to check whether two given code snippets are clones, i.e., equivalent in operational semantics. This dataset is from a broadly recognized benchmark BigCloneBench [2], containing more than six million actual clone pairs and 260,000 false clone pairs from various Java projects. Each data point is a Java method. The defeat detection aims to predict whether a given code snippet contains vulnerabilities. The dataset is extracted from two popular open-sourced C projects: FFmpeg3 and Qemu4. We follow the experimental settings of previous work [15] to use the training set to train the victim models and generate adversarial examples on the test set to evaluate their robustness. For the victim model architecture, we adopt three state-of-the-art (SOTA) pre-trained code language models, including CodeBERT [14], GraphCodeBERT [5], and CodeT5 [6]. The victim model is fine-tuned on each dataset whose details are unknown to the adversary. The statistics and clean accuracy of each victim model are listed in Tab III.

2) Baseline Methods: We adopt MHM [13] and ALERT [15] as the baseline attacks in our cross-domain threat model by replacing their attack objective with the intermediate-level attack objective defined by Eqn. 2. Please note that MHM and ALERT cannot be directly applied to our cross-domain setting because they require extra adversary knowledge. MHM and ALERT utilize identifier renaming as the code transformation operation. To make a fair comparison, we also implement CodeTAE with the identifier renaming only.

3) Hyper-Parameters: We set the hyper-parameters of CodeTAE as follows: In the substitute identifier generation stage of CodeTAE, we set the number of populations as 20, the number of iteration epochs as 10, the number of children generated in each iteration as 10, and the mutation probability as 0.5. In the obfuscated code insertion stage, we control the stealthiness of the generated adversarial examples with the number of changes. Sec. V-D discusses the trade-off between the attack stealthiness and the attack effectiveness of CodeTAE using a single code transformation operation. For the complete CodeTAE combining all the proposed code transformation operations, we set the number of changes as in Tab. IV

B. CodeTAE: Comparing With Baseline Attacks

1) Settings: We compare the proposed CodeTAE with other baseline attacks under the cross-domain threat model under

both the "within the architecture" and the "cross the architecture" settings. The "within architecture" setting denotes that the architecture of the surrogate encoder and the encoder of the victim model are the same. Please note that the "within architecture" setting is still a cross-domain transfer-based setting because the model parameter of the victim model and its classification head is unknown to the adversary. The "cross the architecture" setting denotes that the architecture of the surrogate encoder and the encoder of the victim model are different. We take the CodeBERT pre-trained model as the surrogate encoder to generate adversarial examples. We report the attack results of CodeTAE when using the identifier renaming only for a fair comparison with MHM and ALERT. Please note that the constraint on the stealthiness of CodeTAE is more strict than MHM and ALERT. Specifically, all the identifiers in the code examples can be replaced in MHM and ALERT. In CodeTAE, however, we set the allowed number of changes as two on all three datasets. The experimental results are presented in Tab. VII.

2) Results: We can see that the CodeTAE (identifier renaming only) outperforms the baseline attacks by 42.80%, 2.94%, and 7.50% on each dataset, respectively. The superior attack success rates of CodeTAE (identifier renaming only) illustrate the advantage of the proposed GA-based code obfuscation insertion technique since it is the only difference between CodeTAE (identifier renaming only) and the baseline attacks (MHM & ALERT). Besides, from Tab. VII, we can see that the complete CodeTAE achieves even higher attack success rates compared to CodeTAE (identifier renaming only), illustrating that all the used code transformation operations are effective in enhancing the attack success rates.

3) Discussions: The limitation of CodeTAE is that although it improves the attack success rates on the Clone Detection dataset, its absolute attack success rates are still low on this challenging dataset. Similar low attack success rates on the Clone Detection dataset are also observed in other literature [15]. We explain this phenomenon from the following two aspects. First, we assume that this dataset is challenging to attack because its large-scale data volume makes the feature of the fine-tuned model diverge a lot from that of the pre-trained model, preventing the transfer of adversarial examples. Second, the Clone Detection model may rely more on the global features of the code snippet rather than local features. Thus, the clone detection dataset can be less vulnerable to local perturbations such as identifier renaming and insert attacks. To verify the first explanation, we compare the feature distance between pre-trained models and fine-tuned models on each dataset, and the results are shown in Tab. V. Datasets with higher feature distances are more challenging to attack. The feature distance is calculated as:

$$divergence = \frac{\sum_{i=1}^{n} ||F_s(x_i), F_t(x_i)||_2}{n}, \tag{7}$$

where $F_s(x_i)$ and $F_t(x_i)$ denote the embedding feature vector of code example x_i in the latent feature space of pre-trained model F_s and fine-tuned model F_t, respectively; $||\cdot||_2$ denotes the Euclidean distance, and n denotes the number of examples in the dataset. We can see from Tab. V that the models have the highest feature divergence on the Clone Detection dataset, verifying the first explanation. To verify the second explanation, we calculate the average impact of each identifier on the output confidence score of the victim model and then take an average on the whole dataset as the local sensitivity of this dataset, which is formalized as:

$$sensitivity = \frac{\sum_{i=1}^{n}(\sum_{j=1}^{m_i} I_{i,j}/m_i)}{n}, \tag{8}$$

where n denotes the number of examples in the dataset, m_i denotes the number of identifiers in the $i-th$ example, and $I_{i,j}$ denotes the importance score of the $j-th$ identifier in the $i-th$ example, which is calculated with Eqn. 6. We compare the local sensitivity of victim models on three datasets in Tab. VI, and we can see that the Clone Detection dataset has the lowest local sensitivity, verifying the second explanation.

TABLE V

THE FEATURE DISTANCE BETWEEN PRE-TRAINED ENCODER AND FINE-TUNED ENCODER ON EACH DATASET. THE GCB DENOTES THE GRAPHCODEBERT, AND THE AVG. DENOTES "AVERAGE". THE HIGHEST FEATURE DISTANCES ARE IN **BOLD**

Dataset\Architecture	CodeBERT	GCB	CodeT5	Avg.
Authorship Attribution	599.47	289.19	178.25	355.64
Clone Detection	**907.83**	**649.80**	**269.19**	**608.94**
Defeat Detection	320.14	200.83	107.50	209.49

TABLE VI

THE LOCAL SENSITIVITY OF EACH VICTIM MODEL ON EACH DATASET. THE GCB DENOTES THE GRAPHCODEBERT, AND THE AVG. DENOTES "AVERAGE". THE HIGHEST FEATURE DISTANCES ARE IN **BOLD**

Dataset\Architecture	CodeBERT	GCB	CodeT5	Avg.
Authorship Attribution	0.0101	0.0124	0.0453	0.0226
Clone Detection	**0.0047**	**0.0001**	**0.0021**	**0.0023**
Defeat Detection	0.0091	0.0618	0.0269	0.0326

C. CodeTAE: The Transferability Across Model Architecture

1) Settings: In this experiment, we compare the attack transferability of each single code transformation operation and each surrogate encoder. We refer to Tab.IV to set the number of changes for each code transformation operation. We conduct this experiment on all three datasets using pre-trained CodeBERT, GraphCodeBERT, and CodeT5 as the surrogate encoder, respectively. The results on the Authorship Attribution dataset are presented in Tab. VIII. The results on the other two datasets can be found in the *Appendix*.

2) Results: From the results, we have the following observations. First, the identifier renaming attack is more transferable across different victim models than insert attacks. The average attack success rates of CodeTAE (identifier renaming) are 52.44%, 41.25%, and 45.14% when using different surrogate models, which outperforms other insert attacks by a large margin. Second, insert attacks can be combined with the identifier renaming attack to enhance the attack transferability. Third, larger surrogate models GraphCodeBERT and CodeT5 do not exhibit obvious advantages compared to small surrogate model CodeBERT. The above observations are also

TABLE VII

THE ATTACK SUCCESS RATES (%) OF CODETAE WITH BASELINE ATTACKS MHM & ALERT. THE BEST RESULTS ARE IN **BOLD**. THE SURROGATE MODEL IS PRE-TRAINED CODEBERT FOR ALL THREE DATASETS, AND THE VICTIM MODELS ARE FINE-TUNED CODEBERT, GRAPHCODEBERT, AND CODET5. THIS AVERAGE ATTACK SUCCESS RATE AND OUR IMPROVEMENTS ARE LISTED IN THE LAST TWO COLUMNS

Dataset	Attack	Within Architecture CodeBERT	Cross Architecture GraphCodeBERT	Cross Architecture CodeT5	Average	Our improvements
Authorship Attribution	MHM (Cross-domain variant.)	9.17	8.65	11.11	9.64	\
	ALERT (Cross-domain variant.)	7.48	10.58	8.08	8.71	\
	CodeTAE (identifier renaming only)	43.12	48.54	65.66	52.44	+42.80
	CodeTAE	**59.63**	**58.25**	**76.77**	**64.88**	+55.24
Clone Detection	MHM (Cross-domain variant.)	0.20	0.62	0.41	0.41	\
	ALERT (Cross-domain variant.)	0.41	0.41	0.41	0.41	\
	CodeTAE (identifier renaming only)	5.53	2.05	2.46	3.35	+2.94
	CodeTAE	**6.35**	**4.93**	**2.66**	**4.65**	+4.24
Defeat Detection	MHM (Cross-domain variant.)	9.32	9.27	11.28	9.96	\
	ALERT (Cross-domain variant.)	23.31	12.50	15.95	17.25	\
	CodeTAE (identifier renaming only)	27.12	24.19	22.96	24.76	+7.50
	CodeTAE	**39.83**	**27.02**	**22.57**	**29.81**	+12.55

TABLE VIII

THE ATTACK SUCCESS RATES (%) OF EACH CODE TRANSFORMATION IN CODETAE ON THE AUTHORSHIP ATTRIBUTION DATASET. * DENOTES THE WITHIN-ARCHITECTURE SETTING, AND "COMBINED" DENOTES COMBINING ALL THE PROPOSED CODE TRANSFORMATION OPERATIONS

Surrogate Encoder	Code Transformation	Victim Model CodeBERT	Victim Model GraphCodeBERT	Victim Model CodeT5	Average
CodeBERT	Identifier renaming	43.12*	48.54	65.66	52.44
	Insert unused variable	12.50*	7.69	14.14	11.44
	Insert empty print	6.54*	3.85	9.09	6.49
	Insert empty if	13.08*	10.58	9.09	10.92
	Insert empty while	12.15*	10.58	9.09	10.61
	Combined	59.63*	58.25	76.77	64.88
GraphCodeBERT	Identifier renaming	24.77	50.49*	48.48	41.25
	Insert unused variable	0.92	2.91*	4.04	2.62
	Insert empty print	0.00	0.97*	3.03	1.33
	Insert empty if	0.92	3.88*	3.03	2.61
	Insert empty while	0.92	1.94*	4.04	2.30
	Combined	30.28	66.02*	53.54	49.95
CodeT5	Identifier renaming	23.85	38.83	72.73*	45.14
	Insert unused variable	1.83	6.80	24.24*	10.96
	Insert empty print	1.83	3.88	5.05*	3.59
	Insert empty if	0.92	4.85	11.11*	5.63
	Insert empty while	0.92	3.88	12.12*	5.64
	Combined	23.85	38.83	81.82*	48.17

consistent across the other two datasets Clone Detection and Defeat Detection, as illustrated in the *Appendix*.

D. Trading-off the Stealthiness and the Effectiveness

1) Settings: In this experiment, we study the relationship between the attack success rates of each code transformation and the number of changes. Intuitively, adversarial examples with fewer code perturbations are more stealthy to human observers. We use the number of changes to measure the stealthiness of the adversarial examples generated with each code transformation operation. We study the trading-off between the number of changes and the attack success rates by generating adversarial examples with the pre-trained CodeBERT surrogate encoder and test the attack success rates of the adversarial examples on the fine-tuned CodeBERT classifier. Other attack hyper-parameters are kept the same as in CodeTAE except for the number of changes. Fig. 5 shows the experimental results on the Authorship Attribution dataset. Results on the Clone Detection and the Defeat Detection dataset can be found in *Appendix*. For the identifier renaming attack, the "MAX" denotes renaming all the identifiers in the

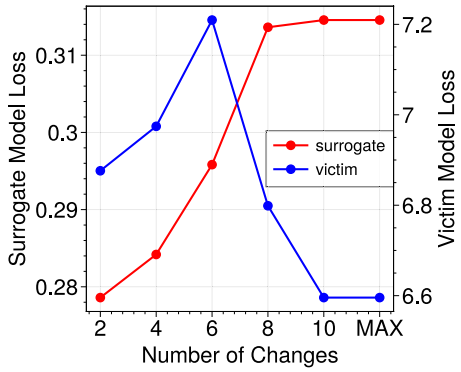


Fig. 4. The loss values of an adversarial example on either the surrogate model (red) or the victim model (blue) under a varied number of changes. We can see that although maximizing the number of changes increases the surrogate loss value, it does not necessarily increase the loss value on the victim model. Using the number of changes less than ten is enough for achieving high loss values on the victim model.

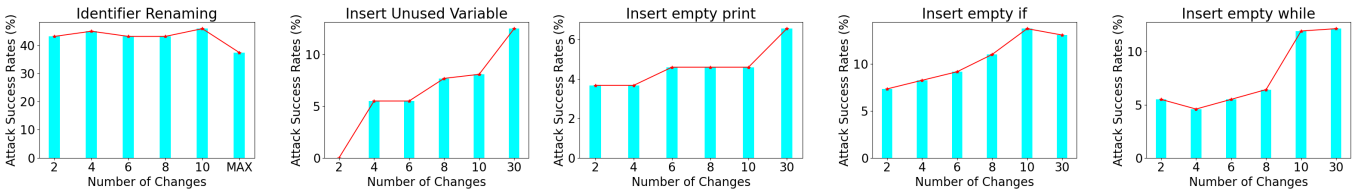


Fig. 5. The attack success rates (%) of each code transformation with a varied number of changes on the Authorship Attribution dataset. The maximal number of changes is 30 for the four insertion attacks. The "MAX" in the figure of identifier renaming attack denotes replacing all the identifiers in a code snippet.

input code. For other insert attacks, we set the maximum number of changes as 30.

2) Results: We can observe that maximizing the number of changes may not lead to the highest attack success rates. Instead, a small number of changes is enough to achieve satisfying results. These results show that CodeTAE can achieve high attack success rates with a limited number of changes, ensuring both the attack's stealthiness and effectiveness.

3) Discussions: We analyze that the above phenomenon is because the feature space of the surrogate encoder and the victim model are different, thus maximizing the surrogate loss by increasing the number of changes may not necessarily lead to the optimal loss on the victim model. We verify the above claim by plotting the loss value of adversarial examples with a varied number of changes. We take the identifier renaming attack and the authorship attribution dataset as an example. We can see from Fig. 4 that although the surrogate loss monotonously increases, the victim loss may not be the highest when maximizing the number of changes.

E. Using CodeTAE in Adversarial Fine-Tuning

1) Settings: In this experiment, we study whether the generated CodeTAE adversarial examples can be used to enhance the generalization and robustness of the victim model. We follow the previous works [15], [16] to generate adversarial examples on the training dataset and then use them to fine-tune the standardly trained victim models. Limited by the computational resources, we only conducted this experiment on the Authorship Attribution and the Defeat Detection dataset. The hyper-parameter setup of the adversarial fine-tuning can be found in the Appendix. After obtaining the adversarially trained models AT-CodeBERT, AT-GraphCodeBERT, and AT-CodeT5, we evaluate their robustness with the attack success rates of adversarial examples transferred from different surrogate encoders. The attack settings of CodeTAE are in Tab. IV

2) Results: The experimental results are presented in Tab. IX. We can see that not only can the CodeTAE adversarial examples enhance the robustness of the code model, but they also benefit the clean accuracy. The clean accuracy increases by 1.25% $\sim$ 6.00%, and the average attack success rates decrease by 8.09% $\sim$ 36.19%. This can be an interesting observation because clean accuracy is often at odds with robustness, and many previous works can only improve the robustness by sacrificing the clean accuracy [41], [53].

3) Discussions: We explain the above phenomenon from the perspective of data diversity and hardness, which was first used in the image domain to explain why data augmentation methods can improve both robustness and generalization [54]. The hardness of the white-box adversarial example is too large for the model to improve generalization. In comparison, adversarial examples with low hardness like corrupted images and transfer-based adversarial examples are beneficial for both robustness and generalization. We assume that the generated CodeTAE transfer-based adversarial examples are low hardness examples, which is similar to the corrupted images. To verify this, we adopt the same technique in [54] to visualize the feature distribution of the generated CodeTAE adversarial examples in Fig. 6. We select the examples of the first three classes in the Authorship Attribution dataset, extract the encoded feature of each example in the embedding space of the victim model, and then visualize the embedding features with the K-means algorithm. The visualization result exhibits a similar pattern as in [54]: Compared to the white-box adversarial examples, the transfer-based black-box CodeTAE adversarial examples can be easily distinguished in the model's feature space, thus improving the data diversity while limiting the training hardness. The high diversity and low hardness of CodeTAE adversarial examples enable the improvement of both robustness and generalization.

VI. Limitations and Future Work

Despite the work of this paper, there are still limitations that deserve to be improved in our future work, including limitations of the evaluation and limitations of the method.

A. Limitations and Future Work of the Evaluation

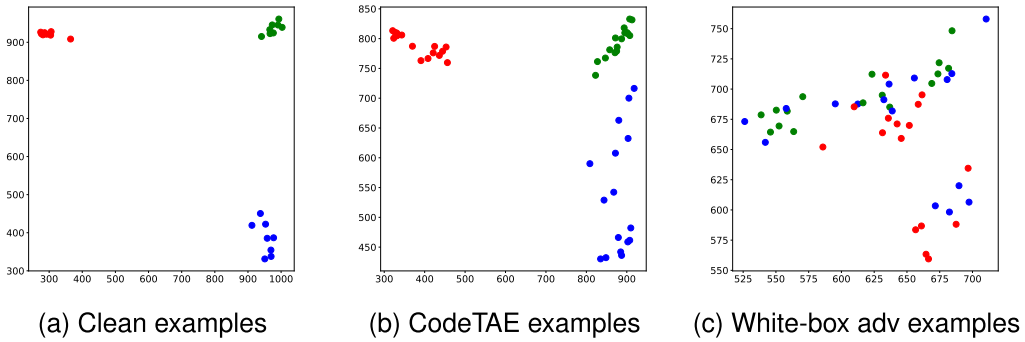
1) The Attack Success Rates Against Code Generation Tasks Deserve to Be Evaluated: This paper only evaluates the attack success rates of CodeTAE against code classification tasks, leaving the evaluation of attack success rates against code generation tasks for future work. Please note that the CodeTAE framework is downstream agnostic, and thus can be directly used to attack the code generation tasks.

2) The Evaluation Metric for the Attack Stealthiness Is Limited: This paper only takes the number of changes as the metric for evaluating the attack stealthiness and assumes adversarial example with a smaller number of changes is stealthier. However, this metric is incomplete because it does not consider the stealthiness differences between each single code transformation. Besides, this paper assumes that shorter substitute identifiers are stealthier for human observation, which may not hold in some cases. As a result, more evaluation metrics for the stealthiness of the generated adversarial code examples deserve to be added.

TABLE IX

THE CLEAN ACCURACY AND ROBUSTNESS (MEASURED BY ATTACK SUCCESS RATE) (%) OF EACH ADVERSARIALLY FINE-TUNED MODEL

Dataset	Model	Clean Acc. ↑	Attack Success Rates ↓ From CodeBERT	From GraphCodeBERT	From CodeT5
Authorship Attribution	CodeBERT	82.58	59.63	30.28	23.85
	AT-CodeBERT	**84.09**	**6.31**	**3.60**	**4.50**
	GraphCodeBERT	79.55	58.25	66.02	38.83
	AT-GraphCodeBERT	**81.06**	**8.41**	**12.15**	**13.08**
	CodeT5	80.30	76.77	53.54	81.82
	AT-CodeT5	**84.09**	**6.31**	**7.21**	**8.11**
Defeat Detection	CodeBERT	60.25	39.83	19.50	19.92
	AT-CodeBERT	**66.25**	**6.42**	**13.96**	**10.94**
	GraphCodeBERT	62.00	27.02	47.52	13.71
	AT-GraphCodeBERT	**63.25**	**6.72**	**14.62**	**7.51**
	CodeT5	64.25	22.57	31.52	46.69
	AT-CodeT5	**68.00**	**9.56**	**21.69**	**23.53**



(a) Clean examples (b) CodeTAE examples (c) White-box adv examples

Fig. 6. Visualization of clean examples, CodeTAE examples, and white-box adversarial (adv) examples in the feature space of victim models. Each dot represents the feature of an example. Examples in different classes are represented with different colors. The CodeTAE adversarial examples are more diverse while not increasing the training difficulty like the white-box adversarial examples.

TABLE X

THE ATTACK SUCCESS RATES (%) OF CODETAE WHEN COMBINING ADDITIONAL THREE CODE TRANSFORMATION OPERATIONS. THIS EXPERIMENT IS CONDUCTED ON THE AUTHORSHIP ATTRIBUTION DATASET WITH A PRE-TRAINED CODEBERT SURROGATE ENCODER

Code Transformation	Victim Model CodeBERT	GraphCodeBERT	CodeT5	Average
unused function variable insertion	7.34	2.86	7.55	5.92
unused function arguments insertion	7.34	13.33	22.64	14.44
comments insertion	9.17	0.95	9.43	6.52

B. Limitations and Future Work of the Approach

1) Combining CodeTAE With More Code Transformation Operations: This paper only discusses the effectiveness of five representative code transformation operations in the CodeTAE framework. The selected five code transformation operations can evaluate the robustness of victim models concerning the data flow, control flow, and API call in the code snippet, respectively. However, CodeTAE is flexible for more code transformation operations. To illustrate this, we additionally evaluate the attack success rates of CodeTAE using three different code transformations, including unused function variable insertion, unused function arguments insertion, and comments insertion. We generate the adversarial examples on the Authorship Attribution dataset using the pre-trained CodeBERT model as the surrogate model. The results are shown in Tab. X. We can see that each code transformation operation achieves a non-trivial attack success rate, illustrating the flexibility of the CodeTAE framework. For further work, we will consider applying more semantic-preserving code transformations in the cross-domain transfer-based attack framework against code models.

2) The Cross-Domain Attack Against Large Language Models (LLMs) Deserves to Be Studied: LLMs have recently achieved tremendous progress in both code understanding and code generation capabilities. However, how to evaluate the robustness of LLMs on source code tasks remains an open question. Compared to the traditional transfer-based attack techniques, the cross-domain attack like CodeTAE is especially suitable for attacking LLMs since it only leverages small models (pre-trained CodeBERT, etc) to craft transferable adversarial examples. We leave this question for future work.

VII. CONCLUSION

This paper proposes a practical cross-domain attack threat model to explore the adversarial example vulnerability of

TABLE XI

THE ATTACK SUCCESS RATES (%) OF EACH CODE TRANSFORMATION IN CODETAE ON THE CLONE DETECTION DATASET USING DIFFERENT SURROGATE ENCODERS. * DENOTES THE WITHIN-ARCHITECTURE SETTING, AND "COMBINED" DENOTES USING ALL THE PROPOSED CODE TRANSFORMATIONS

Surrogate Encoder	Code Transformation	Victim Model CodeBERT	GraphCodeBERT	CodeT5	Average
CodeBERT	Identifier renaming	5.53*	2.05	2.46	3.35
	Insert unused variable	0.61*	0.21	0.20	0.34
	Insert empty print	1.64*	0.41	0.82	0.96
	Insert empty if	0.61*	0.62	0.82	0.68
	Insert empty while	0.61*	0.62	0.82	0.68
	Combined	6.35*	4.93	2.66	4.65
GraphCodeBERT	Identifier renaming	4.92	4.31*	1.23	3.49
	Insert unused variable	0.20	0.00*	0.20	0.13
	Insert empty print	0.41	0.00*	0.41	0.27
	Insert empty if	0.61	0.41*	0.00	0.34
	Insert empty while	0.41	0.41*	0.20	0.34
	Combined	3.89	8.62*	2.66	5.06
CodeT5	Identifier renaming	2.87	2.46	1.84*	2.39
	Insert unused variable	0.82	0.21	0.41*	0.48
	Insert empty print	0.20	0.21	0.00*	0.14
	Insert empty if	1.02	3.08	0.41*	1.50
	Insert empty while	1.02	2.87	0.20*	1.36
	Combined	3.07	4.52	5.42*	4.34

TABLE XII

THE ATTACK SUCCESS RATES (%) OF EACH CODE TRANSFORMATION IN CODETAE ON THE DEFEAT DETECTION DATASET USING DIFFERENT SURROGATE ENCODERS. * DENOTES THE WITHIN-ARCHITECTURE SETTING, AND "COMBINED" DENOTES USING ALL THE PROPOSED CODE TRANSFORMATIONS

Surrogate Encoder	Code Transformation	Victim Model CodeBERT	GraphCodeBERT	CodeT5	Average
CodeBERT	Identifier renaming	27.12*	24.19	22.96	24.76
	Insert unused variable	21.58*	9.27	18.29	16.38
	Insert empty print	17.43*	7.26	9.34	11.34
	Insert empty if	22.41*	8.06	14.79	15.09
	Insert empty while	25.73*	8.87	15.56	16.72
	Combined	39.83*	27.02	22.57	29.81
GraphCodeBERT	Identifier renaming	17.01	31.82*	24.12	24.32
	Insert unused variable	14.94	9.27*	16.73	13.65
	Insert empty print	13.28	8.87*	10.12	10.76
	Insert empty if	15.35	8.87*	14.40	12.87
	Insert empty while	15.77	10.89*	12.84	13.17
	Combined	19.50	47.52*	31.52	32.85
CodeT5	Identifier renaming	4.98	8.06	23.35*	12.13
	Insert unused variable	3.73	3.23	25.68*	10.88
	Insert empty print	3.32	4.03	19.46*	8.94
	Insert empty if	4.56	6.45	31.13*	14.05
	Insert empty while	3.32	4.03	28.40*	11.92
	Combined	19.92	13.71	46.69*	26.77

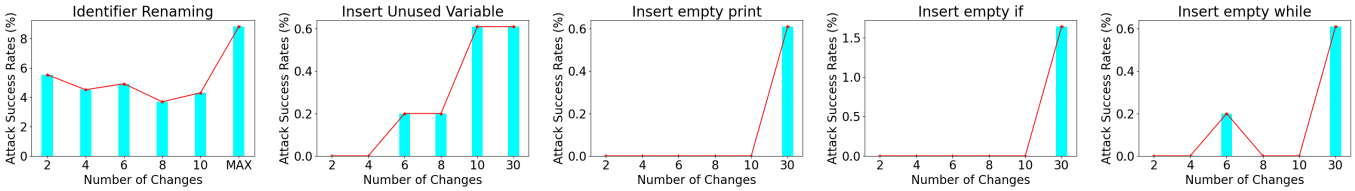


Fig. 7. The Attack success rates (%) of each code transformation with a varied number of changes on the Clone Detection dataset. The maximal number of changes is 30 for the four insertion attacks. The "MAX" in the figure of identifier renaming attack denotes replacing all the identifiers in a code snippet.

transfer learning of source code. Specifically, the proposed CodeTAE attack addresses the attack challenges by utilizing the intermediate-level attack objective and enhancing the transferability of adversarial examples with a genetic algorithm. Extensive experimental evaluations verify the effectiveness of CodeTAE in generating highly transferable and stealthy adversarial examples under the cross-domain setting. Adversarial fine-tuning experimental results further demonstrate the benefits of CodeTAE adversarial examples, contributing to improved robustness and clean accuracy of the source code

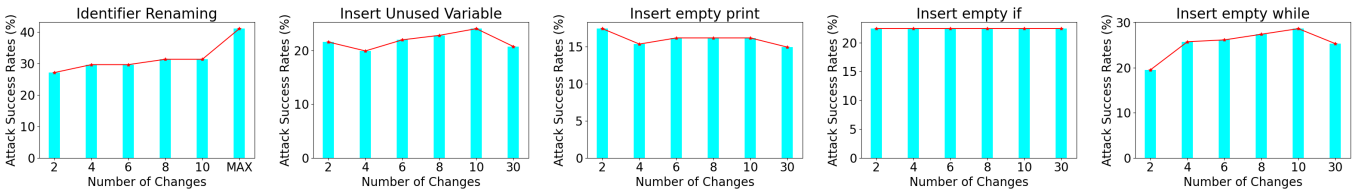


Fig. 8. The Attack success rates (%) of each code transformation with a varied number of changes on the Defeat Detection dataset. The maximal number of changes is 30 for the four insertion attacks. The "MAX" in the figure of identifier renaming attack denotes replacing all the identifiers in a code snippet.

TABLE XIII

COMPARING THE ATTACK SUCCESS RATES (%) OF CODETAE WHEN USING VANILLA GA AND NSGA. THIS EXPERIMENT IS CONDUCTED ON THE AUTHORSHIP ATTRIBUTION DATASET WITH PRE-TRAINED CODEBERT AS THE SURROGATE MODEL. THE SETTINGS OF THE NSGA ARE CONSISTENT WITH THAT OF THE VANILLA GA, AND THE NUMBER OF CHANGES IS TWO FOR ALL CODE TRANSFORMATIONS

Code Transformation	GA variation	Victim Model			Average
		CodeBERT	GraphCodeBERT	CodeT5	
identifier renaming	vanilla	43.12	48.54	65.66	52.44
	NSGA	33.03	32.38	26.42	30.61
insert unused variable	vanilla	12.50	7.69	14.14	11.44
	NSGA	8.26	5.71	7.55	7.17
insert empty if	vanilla	13.08	10.58	9.09	10.92
	NSGA	8.26	5.71	4.72	6.23
insert empty while	vanilla	12.15	10.58	9.09	10.61
	NSGA	7.34	6.67	5.66	6.56
Ensemble	vanilla	59.63	58.25	76.77	64.88
	NSGA	55.05	39.05	36.79	43.63

model. Overall, the design of the CodeTAE attack and corresponding results advance the development of robust intelligent source code processing systems.

APPENDIX

A. CodeTAE Transferability Results on More Datasets

1) Settings & Results: Sec. V-C reports the attack transferability of CodeTAE on the Authorship Attribution dataset. This appendix provides results on two more datasets. The results on the Clone Detection dataset and the Defeat Detection dataset are reported in Tab. XI and Tab. XII, respectively. The attack settings are consistent with Sec. V-C.

2) Discussions: On these two datasets, we have the following observations. From the perspective of code transformation operations, we can see that the identifier renaming transformation is the most effective one, and CodeTAE can achieve even higher attack success rates by combining all the code transformations. From the perspective of surrogate models, we can see that CodeTAE achieves roughly the same attack success rates when using different surrogate models, showing the generalizability of the CodeTAE. Please note that although the absolute magnitude of the attack success rates is not very high, CodeTAE still achieves non-trivial improvements compared to baseline attacks (See Tab. VII). The low attack transferability on the challenging Clone Detection dataset has been discussed in Sec. V-B.

B. Trading-off the Stealthiness/Effectiveness on More Datasets

1) Settings & Results: Sec. V-D have studied the trading-off of CodeTAE between the attack stealthiness and effectiveness. This appendix provides trading-off results on two additional datasets: Clone Detection and Defeat Detection in Fig. 7 and Fig. 8, respectively.

2) Discussions: From the results on these two datasets, we have the following observations. First, maximizing the number of changes may not achieve the optimal transfer attack success rates. Second, CodeTAE can achieve satisfactory attack transferability even with a small number of changes. The above two observations show that CodeTAE can achieve high attack transferability while guaranteeing the stealthiness of the attack, which is consistent with the conclusion on the Authorship Attribution dataset in Sec. V-D.

C. Hyper-Parameters for Adversarial Fine-Tuning

On the Authorship Attribution dataset, we fine-tune the model for 30 epochs with a learning rate of 5e-5 and a batch size of 16. The optimizer is AdamW with $\epsilon = 1e-8$. During fine-tuning, we adopt the gradient clipping strategy with the maximum l_2 norm of 1.0. We adopt the linear learning rate scheduler with warm-up steps as 1/5 of the total training steps. On the Defeat Detection dataset, the hyper-parameters are consistent with the above except for the epochs is 10, the learning rate is 2e-5, the batch size is 32 and the number of warmup steps is 1/10 of the total training steps.

D. Using Other Genetic Algorithms in CodeTAE

CodeTAE uses the vanilla variation of the genetic algorithm (GA), which is single-objective and leverages mutation & crossover operations to generate new populations. We choose the vanilla GA to follow the common practice in attack literature [15], [55], [56], [57], and using other variants may not be better. We have conducted additional experiments to use another variant: the non-dominant sorting genetic

algorithm (NSGA) to attack the victim model. The NSGA is a multi-objective variant of the GA and uses the non-dominate sorting algorithm to select surviving individuals. Formally, the objective function of the vanilla GA can be formalized as:

$$\max_{\delta} ||F(x_{adv}) - F(x)||_2^2, \quad \text{s.t. } Len(T_\delta) < \epsilon \ \& \ x_{adv} = T_\delta(x), \tag{9}$$

where $F(x_{adv})$ denotes the feature embedding of the adversarial example x_{adv} in the latent space of the pre-trained encoder, $||\cdot||_2^2$ denotes the squared L_2 loss, $T_\delta(\cdot)$ denotes the code transformation operation, and $Len(T_\delta) < \epsilon$ is a constraint for the maximum length of the candidate identifier. In comparison, the NSGA can be formalized as:

$$\max_{\delta} ||F(x_{adv}) - F(x)||_2^2, \ \& \ \max_{\delta}\{\min(0, \epsilon - Len(T_\delta))\}, \quad \text{s.t. } x_{adv} = T_\delta(x). \tag{10}$$

The comparison results of vanilla GA and NSGA are shown in Tab. XIII, and we can see that the attack success rates of NSGA are not higher than that of the vanilla GA.

ACKNOWLEDGMENT

The authors would like to thank Weipeng Jiang for his valuable suggestions in improving their research idea and revising their initial manuscript.

REFERENCES

[1] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, "Source code authorship attribution using long short-term memory based networks," in *Proc. ESORICS*, 2017, pp. 65–82.

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

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, Jun. 2019, pp. 4171–4186.

[4] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics EMNLP*, 2020, pp. 1536–1547.

[5] D. Guo, "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. ICLR*, 2020, pp. 1–12.

[6] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder–decoder models for code understanding and generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2021, pp. 8696–8708.

[7] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "CodeT5+: Open code large language models for code understanding and generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2023, pp. 1069–1088.

[8] S. Srikant et al., "Generating adversarial computer programs using optimized obfuscations," in *Proc. ICLR*, 2021, pp. 1–11.

[9] Z. Yu, Y. Wu, N. Zhang, C. Wang, Y. Vorobeychik, and C. Xiao, "Codeipprompt: Intellectual property infringement assessment of code language models," in *Proc. ICML*, 2023, pp. 40373–40389.

[10] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT," 2023, *arXiv:2304.10778*.

[11] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *Proc. USENIX Security*, 2019, pp. 479–496.

[12] Q. Liu, S. Ji, C. Liu, and C. Wu, "A practical black-box attack on source code authorship identification classifiers," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 3620–3633, 2021.

[13] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proc. AAAI*, 2020, pp. 1169–1176.

[14] H. Zhang et al., "CodeBERT-Attack: Adversarial attack against source code deep learning models via pre-trained model," *J. Softw., Evol. Process*, vol. 36, no. 3, p. e2571, Mar. 2024.

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

[16] A. Jha and C. K. Reddy, "CodeAttack: Code-based adversarial attacks for pre-trained programming language models," in *Proc. AAAI*, 2023, pp. 14892–14900.

[17] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2023, pp. 850–862.

[18] Y. Yang et al., "Quantization aware attack: Enhancing transferable adversarial attacks by model quantization," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 3265–3278, 2024.

[19] C. Ma, N. Wang, Q. A. Chen, and C. Shen, "SlowTrack: Increasing the latency of camera-based perception in autonomous driving using adversarial examples," in *Proc. AAAI*, 2024, vol. 38, no. 5, pp. 4062–4070.

[20] C. Szegedy et al., "Intriguing properties of neural networks," in *Proc. ICLR*, 2014, pp. 1–10.

[21] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proc. Asia CCS*, 2017, pp. 506–519.

[22] J. Zheng, C. Lin, J. Sun, Z. Zhao, Q. Li, and C. Shen, "Physical 3D adversarial attacks against monocular depth estimation in autonomous driving," 2024, *arXiv:2403.17301*.

[23] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proc. Int. Conf. Learn. Represent.*, 2015, pp. 1–10.

[24] M. M. Naseer, S. H. Khan, M. H. Khan, F. Shahbaz Khan, and F. Porikli, "Cross-domain transferability of adversarial perturbations," in *Proc. NeurIPS*, 2019, pp. 12905–12915.

[25] Y. Zhang, Y. Song, K. Bai, and Q. Yang, "Cross-domain cross-architecture black-box attacks on fine-tuned models with transferred evolutionary strategies," in *Proc. 31st ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2022, pp. 2661–2670.

[26] Q. Zhang et al., "Beyond ImageNet attack: Towards crafting adversarial examples for black-box domains," in *Proc. ICLR*, 2022, pp. 1–12.

[27] T.-D. Nguyen, Y. Zhou, X. B. D. Le, and D. Lo, "Adversarial attacks on code models with discriminative graph patterns," 2308, *arXiv:2308.11161v1*.

[28] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. NeurIPS*, 2019, pp. 10197–10207.

[29] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. POPL*, vol. 3, 2019, pp. 1–29.

[30] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. ICLR*, 2019, pp. 1–13.

[31] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 5110–5121.

[32] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2020, pp. 1433–1443.

[33] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2020, pp. 473–485.

[34] H. Li et al., "Black-box adversarial example attack towards FCG based Android malware detection under incomplete feature information," in *Proc. USENIX Security*, 2023, pp. 1181–1198.

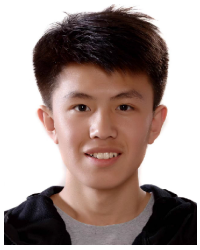
[35] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," in *Proc. OOPSLA*, vol. 4, 2020, pp. 1–30.

[36] J. Jia et al., "ClawSAT: Towards both robust and accurate code models," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2023, pp. 212–223.

[37] Z. Li, X. Huang, Y. Li, and G. Chen, "A comparative study of adversarial training methods for neural models of source code," *Future Gener. Comput. Syst.*, vol. 142, pp. 165–181, May 2023.

[38] F. Gao, Y. Wang, and K. Wang, "Discrete adversarial attack to models of code," in *Proc. ACM Program. Lang.*, vol. 7, 2023, pp. 172–195.
[39] P. Bielik and M. Vechev, "Adversarial robustness for code," in *Proc. ICML*, 2020, pp. 896–907.
[40] J. Henkel, G. Ramakrishnan, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2022, pp. 526–537.
[41] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *Proc. ICLR*, 2018, pp. 1–18.
[42] Y. Wang, M. Alhanahnah, X. Meng, K. Wang, M. Christodorescu, and S. Jha, "Robust learning against relational adversaries," in *Proc. NeurIPS*, 2022, pp. 16246–16260.
[43] Z. Tian, J. Chen, and X. Zhang, "On-the-fly improving performance of deep code models via input denoising," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2023, pp. 560–572.
[44] Z. Li, G. Q. Chen, C. Chen, Y. Zou, and S. Xu, "RoPGen: Towards robust code authorship attribution via automatic coding style transformation," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 1906–1918.
[45] B. Wang, Y. Yao, B. Viswanath, H. Zheng, and B. Y. Zhao, "With great training comes great vulnerability: Practical attacks against transfer learning," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1281–1297.
[46] C. Luo, Q. Lin, W. Xie, B. Wu, J. Xie, and L. Shen, "Frequency-driven imperceptible adversarial attack on semantic similarity," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, New Orleans, LA, USA, Jun. 2022, pp. 15315–15324.
[47] C. Sun et al., "Towards lightweight black-box attack against deep neural networks," in *Proc. NeurIPS*, 2022, pp. 19319–19331.
[48] Q. Zhang, C. Zhang, C. Li, J. Song, and L. Gao, "Practical no-box adversarial attacks with training-free hybrid image transformation," 2022, *arXiv:2203.04607*.
[49] Y. Ban and Y. Dong, "Pre-trained adversarial perturbations," in *Proc. NeurIPS*, 2022, pp. 1196–1209.
[50] J. A. Hernández López, M. Weyssow, J. S. Cuadrado, and H. Sahraoui, "AST-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Oct. 2022, pp. 1–11.
[51] E. Shi et al., "Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2023, pp. 39–51.
[52] Y. Zhi, X. Xie, C. Shen, J. Sun, X. Zhang, and X. Guan, "Seed selection for testing deep neural networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, pp. 1–33, Jan. 2024.
[53] H. Zhang, Y. Yu, J. Jiao, E. Xing, L. El Ghaoui, and M. Jordan, "Theoretically principled trade-off between robustness and accuracy," in *Proc. ICML*, 2019, pp. 7472–7482.
[54] H. Wang, C. Xiao, J. Kossaifi, Z. Yu, A. Anandkumar, and Z. Wang, "AugMax: Adversarial composition of random augmentations for robust training," in *Proc. NeurIPS*, 2021, pp. 237–250.
[55] H. Zhang et al., "Towards robustness of deep program processing models—Detection, estimation, and enhancement," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, pp. 1–40, Jul. 2022.
[56] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 1–18.
[57] L. Wang et al., "DistXplore: Distribution-guided testing for evaluating and enhancing deep learning systems," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2023, pp. 68–80.

Yulong Yang (Graduate Student Member, IEEE) received the B.Eng. degree in computer science and engineering from Xi'an Jiaotong University in 2022, where he is currently pursuing the Ph.D. degree in cyberspace security with the School of Cyber Science and Engineering. His current research interests include adversarial machine learning, model compression, and security risks of multi-modal large models.



Haoran Fan received the B.Eng. degree in software engineering from Dalian University of Technology in 2023. He is currently pursuing the M.Eng. degree in software engineering with the School of Software Engineering, Xi'an Jiaotong University. His current research interests include adversarial machine learning.

Chenhao Lin (Member, IEEE) received the B.Eng. degree in automation from Xi'an Jiaotong University, China, in 2011, the M.Sc. degree in electrical engineering from Columbia University, in 2013, and the Ph.D. degree from The Hong Kong Polytechnic University, in 2018. He is currently a Research Fellow with Xi'an Jiaotong University. His research interests include artificial intelligence security, identity authentication, biometrics, adversarial attack and robustness, and pattern recognition.

Qian Li (Member, IEEE) received the Ph.D. degree in computer science and technology from Xi'an Jiaotong University, China, in 2021. He is currently an Assistant Professor with the School of Cyber Science and Engineering, Xi'an Jiaotong University. His research interests include adversarial deep learning, artificial intelligence security, and optimization of theory.

Zhengyu Zhao (Member, IEEE) received the Ph.D. degree from Radboud University, The Netherlands. He is currently an Associate Professor with Xi'an Jiaotong University, China. His research interests include machine learning security and privacy. Most of his work has concentrated on security (e.g., adversarial examples and data poisoning) and privacy (e.g., membership inference) attacks against deep learning-based computer vision systems.

Chao Shen (Senior Member, IEEE) received the B.S. degree in automation and the Ph.D. degree in control theory and control engineering from Xi'an Jiaotong University, China, in 2007 and 2014, respectively. He is currently a Professor with the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University. His current research interests include AI security, insider/intrusion detection, behavioral biometrics, and measurement/experimental methodology.