

ARTICLE TYPE

CodeBERT-Attack: Adversarial Attack against Source Code Deep Learning Models via Pre-Trained Model

Huangzhao Zhang¹ | Shuai Lu² | Zhuo Li¹ | Zhi Jin^{*1} | Lei Ma³ | Yang Liu⁴ | Ge Li¹¹Key Laboratory of High Confidence Software Technologies, Peking University, Beijing, China²Microsoft Research Asia, Microsoft, Beijing, China³Momentum Lab, University of Alberta, Alberta, Canada⁴School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

Correspondence

*Zhi Jin. No. 1619, No. 1 Science Building. No. 5 Yiheyuan Road, Haidian District, Beijing 100871. Email: zhijin@pku.edu.cn

Abstract

Over the past few years, the software engineering (SE) community has widely employed deep learning (DL) techniques in many source code processing tasks. Similar to other domains like computer vision and natural language processing (NLP), the state-of-the-art DL techniques for source code processing can still suffer from adversarial vulnerability, where minor code perturbations can mislead a DL model's inference. Efficiently detecting such vulnerability to expose the risks at an early stage is an essential step and of great importance for further enhancement. This paper proposes a novel black-box effective and high-quality adversarial attack method, namely CodeBERT-Attack (CBA), based on the powerful large pre-trained model (i.e., CodeBERT) for DL models of source code processing. CBA locates the vulnerable positions through masking and leverages the power of CodeBERT to generate textual preserving perturbations. We turn CodeBERT against DL models and further fine-tuned CodeBERT models for specific downstream tasks, and successfully mislead these victim models to erroneous outputs. In addition, taking the power of CodeBERT, CBA is capable of effectively generating adversarial examples that are less perceptible to programmers. Our in-depth evaluation on two typical source code classification tasks (i.e., functionality classification and code clone detection) against the most widely-adopted LSTM and the powerful fine-tuned CodeBERT models demonstrate the advantages of our proposed technique in terms of both effectiveness and efficiency. Furthermore, our results also show that ❶ pre-training may help CodeBERT gain resilience against perturbations further, and ❷ certain pre-training tasks may be beneficial for adversarial robustness.

KEYWORDS:

Black-box adversarial attack, source code classification, pre-trained model

1 | INTRODUCTION

Over the past years, deep learning (DL) has made great progress and has achieved state-of-the-art performance in multiple application domains, such as image processing, natural language processing (NLP), medical diagnose, etc. In the software engineering (SE) community, we have also been witnessing an increasing trend of leveraging DL techniques for various source code processing tasks. Up to the present, DL models have shown strong capability in achieving state-of-the-art performance for many tasks of source code processing, including functionality classification^{1,2}, code clone detection^{3,4,5}, method naming^{6,7}, code completion^{8,9,10} and code summarization^{11,12,13}. Some of these techniques have further

been developed as applications to increase software development productivity in industry, such as DL-based automated code completion toolkit IntelliCode^{1 14}.

Emerging from representation models, the large pre-trained models have demonstrated great potential in the natural language processing (NLP) community and is later introduced to solve SE tasks. The pre-trained models, such as BERT¹⁵, RoBERTa¹⁶, and GPT series^{17,18,19}, are very large models unsupervisedly trained upon million-level to even trillion-level corpus and provides the basis to achieve state-of-the-art performance for various downstream tasks after fine-tuning. Inspired by the great success of pre-trained models in NLP, CodeBERT²⁰ and later GraphCodeBERT²¹ were proposed for source code processing. Both models are rather complex with 100 million-level parameter size, and are trained upon the CodeSearchNet dataset²², consisting of more than 8.5 million pieces of code data. So far, CodeBERT obtains state-of-the-art performance in tasks from the recently proposed CodeXGLUE benchmark²³, such as natural language code search²² and code defect detection²⁴.

However, the current state-of-the-art DL models still suffer from adversarial vulnerability, where minor perturbations on the inputs can mislead the model's decision. Recently, such non-robust issues of DL models are also reported in the context of source code processing, e.g., when some source code identifiers are substituted, the DL classifiers would fail to produce a consistent prediction²⁵. While verifying how robust a single DL model can be is essential for further enhancement, the adversarial robustness problem for pre-trained models, whose parameter size and corpus size for pre-training are gigantic, and its potential downstream task models, are still largely untouched so far. It is still unclear to what extent state-of-the-art pre-training models for SE tasks are capable to resist adversarial examples.

The adversarial attack, as the process of generating adversarial examples, is a straight-forward approach to verify the robustness of DL models. It probabilistically produces a lower bound of risks (at what probability the model would fail) and an upper bound of robustness (at what probability the model stays robust). A stronger attack will often produce a tighter bound and estimation. Note that DL models embedded in third-party libraries and products are often black-boxes, where only the final outputs of the model are available due to security and privacy constraints. For risk assessment before usage, an effective and efficient black-box adversarial attack approach is of great importance.

In the early attempts to investigate DL vulnerability for source code processing, Metropolis-Hastings modifier (MHM) attack²⁵ is proposed, which carries out iterative random identifier substitution based on the Metropolis-Hastings algorithm^{26,27,28} to generate perturbations. It is able to produce adversarial examples given sufficient budget (time, computational resources, model invocation numbers, etc.), but MHM is not effective and efficient, and the quality of generated examples is also relatively low. All these three drawbacks are caused by the simple searching strategy within the approach. In particular, MHM randomly chooses the identifier to be substituted in a simple way, without searching for the most vulnerable identifier; and randomly chooses the new identifier to substitute to, without considering the quality of the perturbed code.

To address these limitations for more effective and high-quality adversarial attacks, in this paper, we start from a new perspective and take advantage of CodeBERT when performing adversarial perturbations, resulting in a novel black-box adversarial attack for source code processing, namely CodeBERT-Attack (CBA). CBA performs identifier substitution attack iteratively. During each iteration, CBA first locates the vulnerable identifiers in the code by masked classification, which is similar to masked language modeling in CodeBERT. Then, CBA utilizes CodeBERT to generate new perturbed identifier names for the vulnerable identifiers, and at last, CBA probes the victim model with the perturbed examples and selects the one with the lowest predictive probability on the ground truth label. The two major spotlights are: ❶ CBA searches and locates the vulnerabilities, and perform perturbations only upon these positions, and ❷ CBA generates identifier substitutions employing CodeBERT, which is likely to follow the programming language distribution.

To demonstrate the usefulness of our proposed CBA, we perform an in-depth evaluation on two typical classification tasks (i.e., functionality classification and code clone detection), containing about 2 million lines of code, against the most widely-adopted LSTM and the powerful fine-tuned CodeBERT, in black-box scenario. The results show that ❶ CBA generates adversarial examples effectively, which on average reduces the performance of DL models by 31.7%, and in particular, reduces the accuracy of LSTM in OJ from 95.3% to 45.2%; ❷ CBA generates adversarial examples efficiently, as the attack success rate converges quickly, outperforming MHM baseline; ❸ CBA is capable of carrying out successful adversarial attacks against large pre-trained models, as CBA on average reduces the performance of CodeBERT by 17.5%. To the best of our knowledge, this paper is the very first to analyze the adversarial robustness of pre-trained models in the context of source code processing. Our analysis further demonstrates that ❶ pre-training may help the model gain high performance as well as adversarial robustness, as the attack success rate against CodeBERT on average is lower than LSTM by 58.5%; ❷ certain pre-training tasks may enhance the robustness against certain types of adversarial perturbations, as the attack success rate against CodeBERT pre-trained with replaced token detection (RTD, a pre-training task) is lower than CodeBERT pre-trained without RTD by 54.4% on average.

The contributions of this paper are summarized as follow.

¹<https://visualstudio.microsoft.com/services/intellicode/>

²<https://github.com/microsoft/CodeXGLUE>

- We propose a novel black-box adversarial attack method for source code processing, equipped with the powerful pre-trained CodeBERT, namely CBA. Currently, the project is available through the shared link on GitFront³.
- We demonstrate the capability, effectiveness, and efficiency of CBA, against existing state-of-the-art MHM adversarial attack.
- We carry out an adversarial attack against CodeBERT, revealing the non-robust issue of the pre-trained model for source code processing. Besides, we also analyze the robustness enhancement that can be achieved by different pre-training tasks.

2 | RELATED WORK

In this section, we discuss the most relevant works to this paper, including DL for source code processing tasks (Sec. 2.1), pre-trained models (Sec. 2.2), and adversarial attack approaches (Sec. 2.3).

2.1 | DL for Source Code Processing

By far, quite a lot of progress has been made in source code processing by adopting DL techniques in SE community, which can be roughly categorized as classification and generation. In this subsection, we only introduce the most relevant work, and more comprehensive progress along this direction can be referred to the recent survey about big code²⁹.

The DL models for classification tasks make classification predictions based on the vectorized representation of the code. Mou *et al.*¹ and later Zhang *et al.*² propose TBCNN and ASTNN, respectively, for functionality classification. Wei *et al.*³ propose CDLH for code clone detection. Later, the results are improved by PACE⁴ and FA-AST⁵. Weng *et al.*³⁰ and Pradel *et al.*³¹ propose MatchGNet and DeepBugs, respectively, for bug or malware detection.

The DL models for generation tasks take code as input and output the results in sequences, e.g. code token sequences, comment texts, methods names, etc. Allamanis *et al.*⁶ propose convolutional attention networks for method naming, and later the results are improved by Code2Vec⁷. Li *et al.*⁸ propose pointer mixture networks for automated code completion, which is later exceeded by multi-task-learning-based models^{9,10}. Hellendoorn *et al.*³² propose DeepTyper for automated type inference. Hu *et al.*¹¹ propose DeepCom for automated code summarization. The performance is further improved by TL-CodeSum¹² and Code2Seq¹³.

Classification is the basis of generation tasks that can be regarded as a Markov chain of classification. At each generative step that depends on the previous steps, the model selects a token or a word to generate in the manner of classification. In this paper, as an earlier exploratory step, we mostly focus on DL source code classifiers, *i.e.*, functionality classification and code clone detection, and our proposed method can generalize to the generation tasks as well.

2.2 | Pre-Trained Model

The large pre-trained models have demonstrated their unprecedented advantage upon almost every NLP task in the last few years. The pre-trained language models are able to learn contextualized representations of texts by training on a large amount of data, which can then be fine-tuned for specific downstream tasks without training from scratch. Vaswani *et al.*³³ first proposed the transformer architecture with the multi-head attention mechanism, which becomes widely-adopted by the pre-trained models. Emerging from the representation model, Peters *et al.*³⁴ propose ELMo for context-aware word representation. Devlin *et al.*¹⁵ propose BERT for universal bi-directional natural language modeling that predicts a word given the surrounding context. Later, Liu *et al.*¹⁶ propose RoBERTa for robust enhancement of BERT. The GPT series^{17,18,19} are proposed for universal uni-directional natural language modelling, which predicts a word given the prefix sequence. All of the above models have million-level to billion-level parameter size, and are trained on extremely large corpus. In addition, the most recently proposed Switch Transformer model³⁵ has the parameter size at trillion-level, and is trained upon over 100 languages.

Motivated by the huge success of pre-trained models in NLP, SE researchers have also attempted to introduce the pre-training technique into source code processing. Liu *et al.*¹⁰ propose CugLM, incorporating uni- and bi-directional language modeling for pre-training. Karampatsis *et al.*³⁶ propose SCELMO, adopting the ELMo framework to pre-train contextual embeddings for source code. Feng *et al.*²⁰ propose CodeBERT, which is bimodally trained with both natural language and programming language. Guo *et al.*²¹ propose GraphCodeBERT, leveraging DFG of code for programming language modeling. Svyatkovskiy *et al.*¹⁴ propose GPT-C, which is a variant of GPT-2¹⁸ trained from scratch on source code corpus. In this paper, we incorporate CodeBERT into our proposed technique, and also employ the fine-tuned CodeBERT as one of our victim models.

³<https://gitfront.io/r/DrLC5417/5pBsYXgKinB3/CodeBERT-Attack/>

TABLE 1 Summary of notations and symbols in this paper.

Notation	Definition
(x, y)	A pair of code-label example.
$\{t_1, t_2, \dots, t_l\}$	The token sequence of the code x .
$\{s_1, s_2, \dots, s_{l'}\}$	The subtoken sequence of the code x after BPE.
$C_y(x)$	The probability of code x on class y predicted by classifier C .
$l(x)$	The logits of code x produced by classifier C .
$J(y, C(x))$	The instance-level loss function.
$\mathcal{A}(x, y; C)$	The set of adversarial examples of (x, y) against C .
$\mathcal{X}(x, x')$	The indicator of the semantic equivalence of code pairs.
\mathcal{T}	The set of equivalent transformation operators for code.

2.3 | Adversarial Attack

The adversarial attack that generates adversarial examples to uncover the vulnerability of DL models draws much more attention in the past several years. Szegedy *et al.*³⁷ first discover adversarial examples in image classification tasks. Later, Goodfellow *et al.*³⁸, Kurakin *et al.*³⁹, and Papernot *et al.*⁴⁰ propose FGSM, BIM and JSMA, respectively, which are effective and efficient adversarial attack approaches for image processing. Compared to the image task, the adversarial attack of NLP tasks is often more difficult, mostly because that the sentence space is usually discrete, and the similarity of sentences is also relatively hard to measure. Alzantot *et al.*⁴¹ propose GeneticAttack, leveraging genetic algorithm for black-box attack. Ebrahimi *et al.*⁴² propose HotFlip, which performs char-level substitution for white-box attack. Zhang *et al.*⁴³ propose MHA, perturbing words through substitution, insertion, and deletion for both white- and black-box attack. Li *et al.*⁴⁴ propose BERT-Attack, which turns BERT against its fine-tuned models in the manner of black-box attack. Zang *et al.*⁴⁵ propose SememePSO, based on the sememe knowledge base for black-box word substitution.

Besides image processing and NLP domain, researchers also explore adversarial attack for source code processing tasks. The aforementioned challenges in adversarial attack for NLP is even elevated in code space, because programming languages have much more strict lexical, syntactical, and grammatical constraints that an adversarial example must follow. Zhang *et al.*²⁵ first propose MHM for black-box adversarial attack for source code processing, which iteratively performs random renaming of identifiers and leverages reject sampling in the manner of Metropolis-Hastings (M-H) approach^{26,27,28}. Yefet *et al.*⁴⁶ later propose DAMP for white-box attack, which substitutes identifiers and inserts dead code statements according to the gradient guidance. Applis *et al.*⁴⁷ propose LAMPION for the black-box setting, which carries out heuristically defined metamorphic transformations as the perturbation. Srikant *et al.*⁴⁸ define and propose optimized obfuscations for white-box attack. More recently, Yang *et al.*⁴⁹ propose ALERT, for natural black-box attack, which leverages the pre-trained model to produce identifier substitutions. ALERT constraints the pre-trained model by embedding cosine distance to guarantee the naturalness and similarity, which is strong and is likely to fail, therefore ALERT adopts the genetic algorithm as the back-up perturbation. Similar to our proposed CBA, ALERT utilizes pre-trained model for perturbation generation, the attack procedure is different. We regard ALERT as a parallel work and it is quite a good mutual confirmation and complement with our research. Zhang *et al.*⁵⁰ propose a white-box attack approach similar to DAMP, and further integrate it into the robustness detection, estimation and enhancement framework, named CARROT. In this paper, we employ MHM as the baseline for comparison, as MHM is the currently one of the state-of-the-art black-box adversarial attack approach for source code processing.

Besides the adversarial attack approaches mentioned above, the robustness issue of the DL models, especially the pre-trained models, is also a new research direction. Previous work shows the pre-trained language model such BERT may not be robust adversarial attack for NLP tasks^{44,45}. In the field of source code processing, empirical studies^{47,51} also confirms similar conclusions that pre-trained models such as CodeBERT may not be robust against perturbations such as identifier renaming. As the early stage research, we also make an early attempt to investigate the adversarial vulnerability of the pre-trained models in the context of source code processing.

3 | PROBLEM DEFINITION & FORMULATION

As the basis of our proposed technique, in this section, we first provide relevant definitions, formalize the concept of source code classification (Sec. 3.1) and black-box adversarial attack (Sec. 3.2). Then, we give an overview introduction to the large pre-trained models, especially CodeBERT (Sec. 3.3). Table 1 summarizes the notations and symbols used in this paper.

3.1 | Source Code Classification

As discussed earlier, we mostly focus on source code classification tasks in this paper, because they are one of the most basic and typical tasks among all DL applications. For example, source code tasks such as malware detection³⁰ and clone detection^{3,2,4,5} are classification tasks. Other sequential generation tasks, such as code completion^{8,10,9}, code summary^{11,12,13}, and method naming^{6,7}, are essentially a Markov chain of classification, *i.e.* the word or token generated at a certain step is dependent on the previous generations, and each step is a classification to select the word or token to generate.

A well-labeled dataset for source code processing usually consists of a set of source code and their corresponding labels, where each piece of code x is in the form of sequences (character, subtoken or token), trees (AST or complete syntax tree) or graphs (CFG or DFG), *etc.*, according to the model requirements for input format, and y is the label of x . As the basis of DL for source code processing, a source code classifier C can be formulated as a mapping function from the input code to a probability distribution over all possible labels. To be specific, C first encodes the input code $x \in \mathcal{X}$ to extract a feature vector $l(x) = \{l_1(x), \dots, l_{n_l}(x)\}$, which is also known as logits, and then performs softmax classification upon $l(x)$. The overall classification process can be deducted as:

$$C_i(x) = \frac{e^{(W_S l(x) + b_S)_i}}{\sum_{j=1}^{n_c} e^{(W_S l(x) + b_S)_j}}, i = 1, 2, \dots, n_c \quad (1)$$

where n_c is the number of classes, $W_S \in \mathcal{R}^{n_c \times n_l}$ and $b_S \in \mathcal{R}^{n_c}$ are trainable parameters of the softmax layer, and $(\cdot)_i$ takes the i -th element from the vector.

3.2 | Black-Box Adversarial Attack

In this paper, we mainly focus on the black-box scenario of adversary, where the victim model (target model) is treated as a black box. In particular, only the output $C(x)$ of a victim model given x as input can be observed, while other internal behaviors such as structures and weights are invisible. This scenario is quite common in practice, where a large number of DL-based libraries or products are often black-box, which can only be invoked through provided APIs. We formulate black-box attack in the scenario of classification, which can be easily generalized to other more complex generation tasks.

Adversarial example. To define adversarial attack, we first need to introduce the concept of adversarial example. In general, adversarial examples are generated from an input that can be correctly classified by a DL model through some minor perturbations. An adversarial example and its original counterpart are very similar and often indistinguishable from human beings. A set of adversarial examples \mathcal{A} can be generally defined as^{52,53}:

$$\mathcal{A}(x, y; C) = \{\hat{x} | y \neq \arg \max C(\hat{x}) \wedge \|\hat{x} - x\|_p \leq \delta\}, \quad (2)$$

where x and \hat{x} are the original example and the adversarial example, respectively. The first constraint ($y \neq \arg \max C(\hat{x})$) indicates that \hat{x} should mislead C to an erroneous output, and the second constraint ($\|\hat{x} - x\|_p \leq \delta$) limits the perturbation within the allowable L_p distance δ . Note that the formulation is for continuous input spaces such as images. In this paper, we define adversarial examples for source code processing following the previous work²⁵.

Given a well-trained classifier C and a labeled data pair $((x, y))$, where C correctly classifies x to y , *i.e.* $\arg \max C(x) = y$, the adversarial example set \mathcal{A} is defined as:

$$\mathcal{A}(x, y; C) = \{\hat{x} | y \neq \arg \max C(\hat{x}) \wedge \mathcal{X}(x, \hat{x}) = 1\}, \quad (3)$$

where \mathcal{X} is the function determining whether a pair of code is semantically equivalent. To be specific, $\mathcal{X}(x, x') = 1$ when x and x' are compilable, and for any legal input, the executional results of x and x' are consistent; otherwise, x and x' are incompilable or inequivalent, and $\mathcal{X}(x, x') = 0$. There are also two constraints in Eq. 3. The first constraint is the same as Eq. 2, which aims to mislead C , and the second constraint ($\mathcal{X}(x, \hat{x}) = 1$) ensures that \hat{x} is semantic equivalent with x , from the perspective of compilation and runtime execution.

Note that the definition of adversarial examples for source code processing differs from other fields such as NLP in the second constraint. As in NLP, the generated examples must be natural and have similar meanings to the original ones. The previous work²⁵ substitutes the meaning constraint with the compilation constraint, because the similarity of two pieces of code is difficult to measure. Although Eq. 3 does not require the similar meaning of code, we still seek to preserve the textual meaning during perturbation, because the generated examples should fit the distribution of programming languages and have *high quality*.

Adversarial attack. Adversarial attack is the process to generate adversarial examples from the original inputs. A widely adopted idea is to regard adversarial attack as an optimization problem. There are two types of settings for adversarial attack – targeted attack and untargeted attack. Targeted attack aims to mislead C from the ground-true y to a certain $\hat{y} (\hat{y} \neq y)$. In this paper, we focus on untargeted attack, where any $\hat{y} \neq y$ is feasible. In the general form, the optimization objective can be defined as:

$$\max_{\hat{x}} J(y, C(\hat{x})), \text{ s.t. } \|\hat{x} - x\|_p \leq \delta, \quad (4)$$

where J is the loss function, which shows the impact of the perturbation. In practice, the optimization process terminates when an adversarial example is found, without having to exactly reach the global optima. Therefore, the less effective attack may miss the adversarial examples. When coming to source code, we also substitute the constraint in Eq. 4 with the second constraint in Eq. 3. Hence, optimization objective of adversarial attack for source code processing can be formulated as:

$$\max_{\hat{x}} J(y, C(\hat{x})), \text{ s.t. } \mathcal{X}(x, \hat{x}) = 1, \quad (5)$$

However, proving the semantic equivalence of two code snippets is theoretically undecidable, and it is often impractical to check the consistency of the executional results on all possible inputs, whose amount is often quite large or even infinite. Therefore, we simulate \mathcal{X} with equivalent transformation operators $t \in \mathcal{T}$ (e.g. identifier substitution), where $\mathcal{X}(x, t(x)) = 1$. Transformation of x for k times, denoted as $t^k(x)$, still satisfies $\mathcal{X}(x, t^k(x)) = 1$. Therefore, adversarial attack for source code processing can be further described as a combinatorial optimization problem, where the attacker aims to find a proper combination of $t_{i_1}, \dots, t_{i_m} \in \mathcal{T}$ that produces an adversarial example.

Previously proposed MHM. In order to solve the combinatorial optimization problem, Zhang *et al.*²⁵ propose a searching-based sampling approach, namely Metropolis-Hastings modifier (MHM), which is the current state-of-the-art black-box adversarial attack algorithm against code classifiers. MHM regards the problem as a sampling problem. The stationary distribution π is formulated as:

$$\pi(x') \propto (1 - C_y(x')) \cdot \mathcal{X}(x, x'), \quad (6)$$

This formulation is similar to the optimization problem (Eq. 5), because sampling from Eq. 6 tends to sample examples with low ground-truth probability ($C_y(x')$), which finally leads to a large loss $J(y, C(x'))$.

MHM iteratively substitute identifiers in the code, and each iteration consists of two phases. The first phase generates a proposal to rename an identifier t to t' , which are uniformly sampled, forming x' , and the second phase accepts or rejects the proposal ($x \rightarrow x'$) according to the acceptance rate α computed in the manner of Metropolis-Hastings^{26,27,28}, formulated as:

$$\alpha = \min\{1, \alpha^*\} \approx \min\left\{1, \frac{1 - C_y(x')}{1 - C_y(x)}\right\} \quad (7)$$

The major drawbacks of MHM are caused by randomness. From the perspective of searching, MHM makes simple random attempts at each iteration, which is neither effective nor efficient. Another possible issue is that the generated adversarial examples can be “unnatural”. The identifier substitutions are completely randomized, resulting in code that does not conform to programmers’ coding habits. Such examples can in fact be easily distinguished by programmers, whose quality is unsatisfactory. In this paper, we tackle both problems caused by randomness by ❶ searching for vulnerability in code, which ensures that CBA does not waste resources upon the ineffective perturbations, and ❷ employing CodeBERT to produce candidate perturbations, which satisfy the programming language distribution with high quality.

3.3 | Pre-Trained Model

With the rapid development of DL, the ways of applying neural networks have also changed greatly. Recently, with the emergence of the powerful transformer architecture and the large pre-training corpus, instead of training independent end-to-end models solely from scratch, researchers have managed to develop powerful pre-trained models following the “pre-training and fine-tuning” paradigm. The pre-trained models, stacking multiple transformer layers (often), are pre-trained upon extremely large corpora to acquire general knowledge about the language, and then, researchers may further fine-tune the models on the downstream tasks based on the pre-trained parameters. Pre-trained models have been proven to be capable of learning universal language representations in the field of NLP^{34,17,18,15,19}. Later, pre-trained models are also introduced into the field of SE, resulting in CodeBERT²⁰ and GraphCodeBERT²¹. Both pre-trained models have shown their great capability to promote the performance of SE tasks.

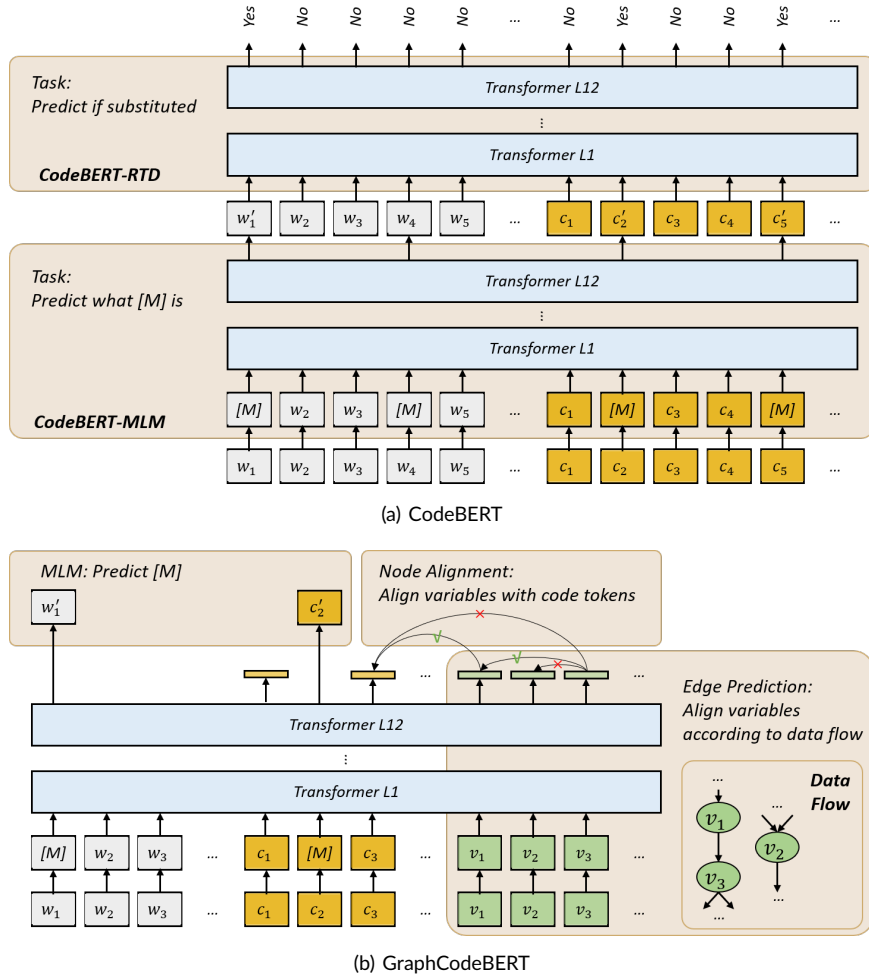


FIGURE 1 Illustrative examples of CodeBERT and GraphCodeBERT. Both the models consists of 12 transformer layers (L1-L12). (a) CodeBERT takes source code (c_1, c_2, \dots) paired with (optional) comment (w_1, w_2, \dots) as input, and is pre-trained by MLM and RTD tasks. Specifically, the two sub-models (CodeBERT-MLM and CodeBERT-RTD) are pre-trained in a GAN-style. MLM aims to recovery the masked tokens and RTD aim to detect whether each token is substituted. (b) Besides the code-comment pair, GraphCodeBERT takes data flow into consideration, by feeding the additional variable sequence (v_1, v_2, \dots) into the model. Two more pre-training tasks are introduced by GraphCodeBERT – node alignment aligns the variable with the code token, and edge prediction learns “where the value comes from” according to the data flow.

We introduce CodeBERT²⁰ and GraphCodeBERT²¹ in this subsection, and we employ them as subject models in our experiments. In this paper, we incorporate CodeBERT²⁰, which is a bimodal pre-trained model for natural language and programming language, into the adversarial attack, forming CBA. Based on the programming language modeling by CodeBERT, CBA is even able to mislead the fine-tuned CodeBERT and GraphCodeBERT for downstream classification tasks.

Architecture & corpus. The architecture of CodeBERT and GraphCodeBERT (illustrated in Fig. 1) is based on RoBERTa¹⁶, which is a pre-trained model proposed in the field of NLP. Both CodeBERT and GraphCodeBERT employ the transformer architecture³³ as the backbone. To be specific, they consist of 12 transformer layers (L1-L12), and each layer has a hidden size of 768 and 12 attention heads. The overall parameter size of each model is over 125 million. The models are trained upon the CodeSearchNet dataset²², which consists of publicly available open-source non-fork GitHub repositories in six programming languages (*i.e.*, Go, Java, JavaScript, PHP, Python and Ruby). The training corpus includes both bimodal and unimodal data, where a piece of bimodal data consists of a source code function and its corresponding documentation description, while a piece of unimodal data contains the function only. In particular, there are 2.1 million function-documentation pairs as bimodal data and 6.4 million unimodal functions for training. In short words, the model size and the training data corpus size are much larger than the classic downstream tasks in SE community.

Adaptation. To adapt to downstream SE tasks, CodeBERT and GraphCodeBERT usually serve as representation models, *i.e.*, they encode the token sequence into a vectorized representation. Specifically, for classification tasks, which are the subject tasks studied in this paper, CodeBERT and GraphCodeBERT insert a special <CLS> token into the very beginning of the input sequence, and make the prediction with multiple dense layers based on the contextual representation of <CLS>. The pre-trained model is fine-tuned (further trained) upon the downstream datasets, such as clone detection. Besides fine-tuning, there is another new adaptation approach of the pre-trained model, namely prompting⁵⁴. Prompt learning is a light and efficient paradigm to apply the pre-trained model into the downstream tasks (usually small datasets or even few-/zero-shot). Prompting directly adds an additional sequence (manually designed or automatically learned) to the token sequence, and attempt to induce the knowledge learned by the pre-trained model. *E.g.*, in a bug detection task, we can feed a sequence of the source code along with an additional template, such as “Is this code snippet buggy? <MASK>”, into the pre-trained CodeBERT, and the model may produce buggy-or-not prediction at <MASK> without fine-tuning. Please refer to the good survey for more details of prompt⁵⁴. In our paper, we do not involve design of the DL model. Therefore, we respect the existing usage of the models by following the fine-tuning procedure.

CodeBERT. CodeBERT is pre-trained by two tasks – masked language modeling (MLM) and replaced token detection (RTD). MLM, introduced by BERT¹⁵, randomly masks tokens with the special token <MASK> and forces the model to predict what <MASK> is. On the other hand, RTD, proposed by ELECTRA⁵⁵, aims to detect which token in the given sequence is substituted. MLM and RTD work as adversary in CodeBERT, leading to two sub-models – CodeBERT-MLM and CodeBERT-RTD, as illustrated in Fig. 1(a). The goal of CodeBERT-MLM is to fool CodeBERT-RTD by revocrying the masked sequence, and CodeBERT-RTD’s purpose is to detect replacements. Pre-trained in a GAN-style⁵⁶, the two sub-models compete with each other and improve each other. Currently, there are two released versions of CodeBERT: CodeBERT obtained through RTD⁴ (where CodeBERT-MLM is discarded after the pre-training is complete), and CodeBERT-MLM obtained through MLM only⁵. In this paper, we employ CodeBERT-MLM to carry out CBA attack, and we leverage our attack against both CodeBERT models to examine the capacity of our proposed approach and the adversarial vulnerability of the pre-trained models.

GraphCodeBERT. As shown in Fig. 1(b), apart from the code-comment pair, GraphCodeBERT also takes the variable sequence as input. GraphCodeBERT incorporates data flow knowledge into pre-training, by introducing two tasks – node alignment and edge prediction. ❶ Node alignment requires the model to align each variable with its corresponding appearance in the code token sequence. This pre-training task helps GraphCodeBERT to align nodes in the data flow with the code snippet. ❷ Edge prediction demands the model to learn “where the value of each variable comes from”. *E.g.*, in the demonstrative example in Fig. 1(b), the value of v_3 comes from v_1 rather than v_2 , according to the data flow. Therefore, the edge prediction between v_1 and v_3 is true, while v_2 and v_3 false. This pre-training task enables GraphCodeBERT to learn from data flow, and introduces data flow knowledge into the pre-trained model. We finetune the released GraphCodeBERT⁶ and evaluate CBA against it in our paper.

4 | CodeBERT-Attack

In this section, we first present a brief overview of our proposed CBA (Sec. 4.1). Then, we illustrate our internal design of CBA in detail (Sec. 4.2 & 4.3). After that, we elaborate key know-hows we explore and adopt in our design and implementation (Sec. 4.4).

4.1 | Overview

We first give a high-level description of our newly proposed method CodeBERT-Attack (CBA) that utilizes the CodeBERT language model to generate adversarial examples to mislead the fine-tuned CodeBERT models or other DL models. Our approach iteratively performs black-box identifier substitution, and each iteration consists of three steps: ❶ locating the vulnerable identifiers of victim model, ❷ generating candidate substitutions with CodeBERT, and ❸ probing the victim model with the potential adversarial examples. CBA repeats these steps until an adversarial example is found, or the allocated resource (*e.g.* iterations or computational time) exhausts. Algo. 1 summarizes the key workflow of CBA.

To be specific, in step one, we locate the pivotal identifiers that are important to the classification of the victim model by masked classification (Fig. 2(a)). The more important these identifiers are, the more likely the perturbations upon them would form an adversarial attack. Then, in step two, we generate the candidate substitutions by choosing the top- k possible combinations of subtokens predicted by masked language modeling employing CodeBERT (Fig. 2(b)). Then, we test the candidates against the victim model, to determine whether to terminate (detecting an adversarial example or exceeding the allocated budget). In the following iteration, we select the substitution that causes the largest decrease in the probability

⁴<https://huggingface.co/microsoft/codebert-base>

⁵<https://huggingface.co/microsoft/codebert-base-mlm>

⁶<https://huggingface.co/microsoft/graphcodebert-base>

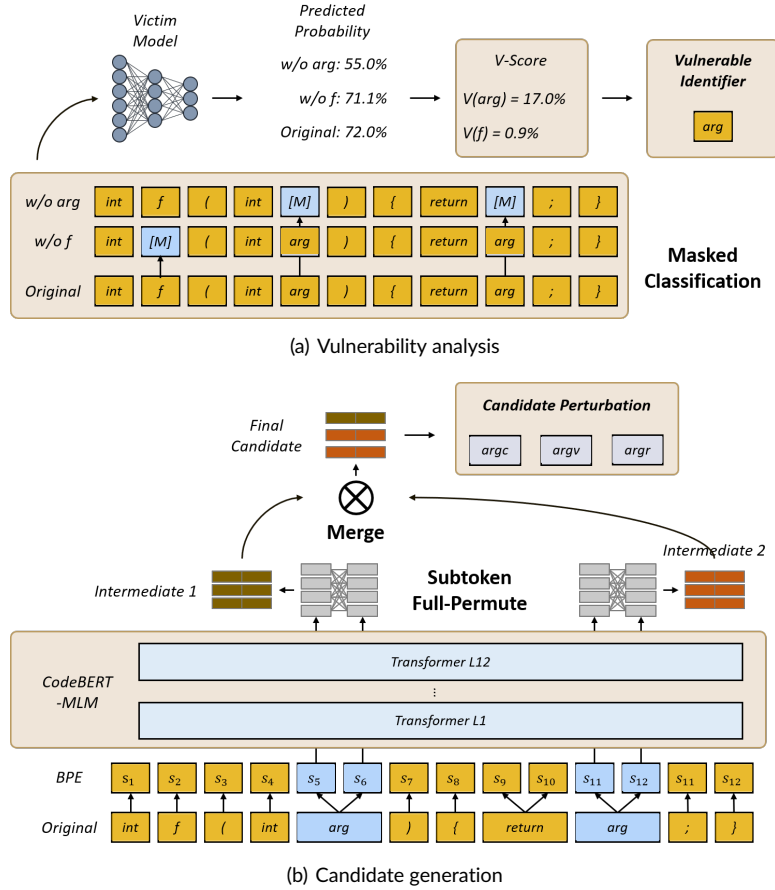


FIGURE 2 An illustrative example of our proposed CBA. (a) shows the process to locate vulnerable identifiers (with high V-score) by masked classification. [M] refers to the masked identifier. (b) presents the process to generate candidate substitutions of vulnerable identifiers. In this specific example, the vulnerable variable “arg” appears twice in the code, and therefore, CBA generates two intermediate candidate sets for each appearance of “arg” (see “Subtoken Full-Permute” in the figure). Then, CBA takes both location into consideration and merges these two intermediate sets, forming the final candidate set (see “Merge” in the figure). Later, for each candidate, CBA replaces the “arg” tokens with the specific candidate, leading to a perturbed code example.

of the ground-truth label predicted by the victim model. If there does not exist such examples, CBA also terminates itself to save time and resources for other trials (Line 18-21 in Algo. 1).

4.2 | Vulnerability Analysis

As discussed earlier, perturbations upon simple random of arbitrary positions are neither effective nor efficient. In this paper, we propose vulnerability analysis in CBA for further guidance of attacks. Intuitively, if a prediction of the victim model highly depends on a particular identifier, this identifier is often likely to be vulnerable, as some minor perturbations upon this identifier would easily break the dependency and eventually fail the victim model. In other words, to locate vulnerable identifiers is to identify the “important” ones for a decision of the model. Inspired by rational analysis^{57,58}, we adopt masked classification to determine whether and to what extent an identifier is vulnerable (Line 5 in Algo. 1).

Give a code token sequence x , CBA retrieves the user-defined identifiers (e.g., functions and variables defined and used within the snippet), and builds a set of identifier masked sequence, denoted as $\{m^1(x), m^2(x), \dots\}$. Specifically, for the i -th identifier, CBA builds $m^i(x)$ by replace all appearances of the i -th identifier with the mask token <MASK>. For instance, in Fig. 2(a), the original x contains two identifiers – “f” and “arg”, and CBA builds two $m^i(x)$ by masking “f” and “arg” respectively, i.e. $m^1(x) = \text{“int <MASK> (int arg) return arg;”}$ and $m^x(x) = \text{“int f (int <MASK>) return <MASK>;”}$. Then we feed each $m^i(x)$ into the victim model, obtaining the V-score (vulnerability-score) of each identifier. For $m^i(x)$ with the i -th identifier in x masked, the V-score is formulated as the probability decrease on the ground-truth class y predicted by the victim model C :

Algorithm 1 CodeBERT-Attack Algorithm.**Inputs:** Victim classifier C , code-label pair (x, y) **Outputs:** Adversarial example $\hat{x} \in \mathcal{A}(x, y; C)$

```

1:  $x_0 \leftarrow x$ 
2: for  $i \in \{1, 2, \dots, \text{ITERATION\_BUDGET}\}$  do
3:   if  $y \neq \arg \max C(x_{i-1})$  then return  $x_{i-1}$ 
4:   end if
5:    $v_1, \dots, v_{n_{vul}} \leftarrow \text{MASK\_CLS}(x, y, n_{vul})$  ▷ Select top  $n_{vul}$  vulnerable identifiers by masked classification
6:    $c_1, \dots, c_{l'} \leftarrow \text{CodeBERT\_MLM}(x_{i-1})$  ▷ Obtain subtoken-level probability sequence by Codebert
7:    $\mathcal{C} \leftarrow \{\}$ 
8:   for  $j \in \{1, \dots, n_{vul}\}$  do
9:      $\mathcal{L}_1, \dots, \mathcal{L}_m \leftarrow \text{LOCATE}(x, v_j)$  ▷ Locate the vulnerable identifier in the token sequence
10:    for  $k \in \{1, \dots, m\}$  do
11:       $s_{i'}, \dots, s_{j'} \leftarrow \text{ALIGN\_SUBTOKENS}(x, \mathcal{L}_k)$  ▷ Align the token sequence with the subtoken sequence
12:       $S_k \leftarrow \text{PERMUTE\_AND\_TOP}(\{c_i\}_{i=i'}^{j'}, n_{inter})$  ▷ Generate the intermediate candidate set by top  $n_{inter}$  permutation
13:    end for
14:     $\mathcal{S} \leftarrow \text{MERGE}(\{S_k\}_{k=1}^m)$  ▷ Merge the intermediate candidate sets from different locations
15:     $\mathcal{C} \leftarrow \mathcal{C} \cup \text{TOP}(\mathcal{S}, n_{cand})$  ▷ Select top  $n_{cand}$  from the merged set forming the candidate set
16:  end for
17:   $\mathcal{X}' \leftarrow \text{SUBSTITUTE}(x, \mathcal{C})$  ▷ Substitute the vulnerable identifiers according to the candidate set
18:   $x' \leftarrow \arg \min_{x' \in \mathcal{X}'} C_y(x')$  ▷ Probe the victim model with the perturbed examples
19:  if  $C_y(x') \geq C_y(x_{i-1})$  then return NONE
20:  end if
21:   $x_i \leftarrow x'$ 
22: end for return NONE

```

$$V(x, i; C, y) = C_y(m^i(x)) - C_y(x) \quad (8)$$

The V-score approximates to what extent a prediction of C depending on an identifier, further indicating the vulnerable level. In general, the higher the V-score is, the more probable perturbations upon this identifier would mislead C . In CBA, we select the top n_{vul} identifiers with the highest V-score as the vulnerable identifiers, where n_{vul} is the hyper-parameter of vulnerable identifier number.

4.3 | Candidate Generation

Instead of simple random generation, CBA utilizes CodeBERT to generate substitution candidates for the vulnerable identifiers, which brings a major advantages: CodeBERT can produce candidates with high quality, as they are likely to satisfy the distribution of programming languages. Thanks to vulnerability analysis in the previous step, CBA precisely applies small and natural perturbations at the vulnerable position, boosting the effectiveness and efficiency of adversarial attack.

In particular, CBA feeds the BPE-splitted subtoken sequence $\{s_1, \dots, s_{l'}\}$ into CodeBERT without masking, obtaining the subtoken-level prediction probability sequence $C(x) = \{c_1, \dots, c_{l'}\}$, where each c_i is a probability vector over the subtoken vocabulary at position i predicted by CodeBERT (Line 6 in Algo. 1). The reason we do not mask the identifiers is that we aim to preserve the textual meaning after perturbation⁵⁹. According to the previous study, when employing BERT-like model for synonymous word substitution, masking makes the pre-trained MLM model to substitute the target word completely based on the context, which can be reasonable, but completely unrelated to the original word. For instance, given a sequence “I like the dog”, if we mask the word “dog”, almost any noun could be an appropriate substitution based on the context, such as “weather” or “movie”, but they are not related to the original word “dog”. However, if we keep the word, besides the context, the pre-trained MLM model also needs to consider the textual information of “dog”, leading to possible substitutions such as “puppy”. In addition, unmasking makes the computational cost rather low, compared with masking. Without masking, in each iteration, CBA invokes CodeBERT-MLM only once (Line 6 in Algo. 1). The substitution candidates are generated based on the predicted probability sequence by CodeBERT-MLM. However, if mask, we have to build masked sequences for each vulnerable identifier and call CodeBERT multiple times, which is much more costly.

Then, CBA generates the candidate substitution in two phases. In phase ❶, CBA builds the intermediate candidate set at each appearance position of the identifier to be substituted in x . Suppose the identifier appears at position \mathcal{L}_i , corresponding to token t_i , and t_i is split into $\{s_{i'}, \dots, s_{j'}\}$ after BPE, $\{c_{i'}, \dots, c_{j'}\}$ would result in CodeBERT's prediction for t_i . Ideally, we select the subtoken combinations with the highest probability as the desired intermediate candidate set. However, this can often be difficult and even impractical, because the computation of probabilities of all subtoken combinations is a series of chain multiplications over $\{c_{i'}, \dots, c_{j'}\}$, which causes exponential explosions. Therefore, instead of considering all possibilities, we approximately consider only the subtokens with high probabilities at each position. For each subtoken $c_j \in \{c_{i'}, \dots, c_{j'}\}$, CBA selects the top- k subtokens according to c_j , and computes the probabilities of combinations, i.e., the full-permutation within the selected subsets (as shown in "Subtoken Full-Permute" in Fig. 2(b)). In other words, in this phase, CBA employs the pre-trained CodeBERT to model the marginal distribution of each substitution for each location independently. CBA generates the intermediate candidate set for t_i by choosing the top- k combinations with highest probabilities (Line 12). Since an identifier may appear multiple times in x , we have to collect the most probable substitutions at each position first. In the demonstrative example of Fig. 2(b), the vulnerable variable "arg" appears twice in the code, and CBA produces two intermediate sets correspondingly.

Now we have several intermediate candidate sets, and each set refers to the substitution recommended by CodeBERT at a specific position where the identifier appears. Phase ❷ merges these intermediate sets, and form one set of subtoken combinations for identifier substitution, denoted as \mathcal{S} . Suppose the intermediate sets are S_1, \dots, S_m , when the identifier appears m times in x , the probability of the subtoken combination \mathcal{I} , which would finally concatenate as a legal token, can be simply deducted as:

$$\text{prob}(\mathcal{I}) = \begin{cases} 0, & \text{if } \mathcal{I} \text{ is illegal} \\ \prod_{i=1}^m \text{prob}_i(\mathcal{I}), & \text{otherwise} \end{cases}, \quad (9)$$

where $\text{prob}_i(\mathcal{I})$ is the probability of \mathcal{I} in intermediate set S_i . When \mathcal{I} is illegal, or \mathcal{I} is not in one of the intermediate sets, $\text{prob}(\mathcal{I}) = 0$. Note there are three types of illegal \mathcal{I} listed as below. ❶ \mathcal{I} does not comply with the naming rules of the certain programming language. E.g., in most programming languages, a valid identifier can have letters (uppercase "A"-"Z" and lowercase "a"-"z"), digits ("0"-"9") and underlines ("_"), with the further constraint that the very first letter must be either a letter or an underline. Therefore, identifiers such as "1index" is illegal. ❷ \mathcal{I} is in the stop-word set. This set consists of key words (e.g., "for" and "while") and commonly used library functions (e.g., "printf" and "scanf" from "stdio.h" in the C/C++ language). Additionally, "main" is also included in the stop-word set. If \mathcal{I} is in the stop-word set, it is not likely to be a user-defined identifier. For compilability considerations, we consider all stop-words as illegal. ❸ \mathcal{I} is not a legal token. This is caused by BPE in CodeBERT. BPE employs a special letter " \dot{G} " to indicate the beginning of a new token. For instance, " \dot{G} il" refers to a new token with the beginning subtoken of "il", and the following subtoken "legal" makes the rest of the whole token "illegal". If \mathcal{I} does not begin with " \dot{G} " (e.g., "index") or \mathcal{I} contains " \dot{G} " in the middle (e.g., "in \dot{G} dex"), the subtokens in \mathcal{I} cannot be synthesized into a token, and therefore it is illegal.

Similarly, CBA selects top- k subtoken combinations with high merged probabilities from the merged set, forming the final candidate set for substitution of this identifier (Line 14-15). When configuring with different k , we adjust the selection scope of tentative candidates. From the perspective of probability distribution, this phase merges the marginals to build the joint distribution, considering each position where the vulnerable identifier appears. Take the demonstrative example in Fig. 2(b) for instance, by merging the two intermediate sets into a single one, CBA selects the top plausible substitution candidates of "arg", i.e., "argc", "argv", and "argr". Later, CBA probes the victim model with these candidates and completes the whole iteration.

Note that there may be inappropriate substitution candidates produced during the two phases, such as illegal identifier and remaining identical to the original one. After the intermediate sets and the final candidate set are built, CBA performs a filtration, removing all illegal identifiers and the original one (if it appears in the set), leaving the rest to the next step.

4.4 | Key Know-hows of CBA

In this part, we describe some key know-how we explore and gain during the design and implementation of CBA.

MLM. As demonstrated in Fig. 2(b), CBA only needs CodeBERT-MLM to predict subtoken probability at specific positions where the vulnerable variable appears. Ideally, we may modify the final (the 12-th) transformer layer in CodeBERT to compute only upon such positions. However, it takes much engineering effort under the current mainstream framework such as the Hugging Face Transformers package. So CBA takes a step back, and employs a much easier implementation without losing much computational efficiency – we allow CodeBERT to compute subtoken probabilities over the entire sequence, and collect only the target subtokens for further computation. Due to the self-attention mechanism in the transformer architecture, CodeBERT has to compute over the entire subtoken sequence in its lower 11 layers (note that CodeBERT consists of 12 transformer layers), in order to capture the contextual information. Assuming the subtoken sequence is long and the vulnerability appears sparsely (only a few

times), the computational cost of the 12-th layer in the ideal implementation can be neglected, and the overhead of our implementation is less than $\frac{1}{12}$ compared with the ideal one, which is acceptable for modern GPUs in our case.

Smoothing. According to Eq. 9, $\text{prob}(\mathcal{I}) = 0$ if $\exists i$, s.t. $\mathcal{I} \notin \mathcal{S}_i$. However, only few combinations are in all intermediate sets, *i.e.*, the merging multiplications often produce zero probabilities. To tackle this issue, we employ the smoothing technique in the merging process. When \mathcal{I} is not in \mathcal{S}_i , we assign $\text{prob}_i(\mathcal{I})$ with:

$$\text{prob}_i(\mathcal{I}) = \beta \cdot \min_{\mathcal{I}' \in \mathcal{S}_i} \text{prob}_i(\mathcal{I}') \quad (10)$$

which is a very small non-zero value, so that subtoken combinations are allowed to be absent in some intermediate sets. β is a hyper-parameter, and we fix it to 0.1.

Cut-off. Although CBA adopts the approximation by only considering the top- k subtokens for each $c_j \in \{c_{j'}, \dots, c_{j''}\}$, the computational cost is still exponential. Therefore, we only consider the very first intermediate candidate sets during merging for the identifier when a computational budget exhausts, regardless of the rest. We regard this budget as a hyper-parameter of CBA.

Language model filtration. Algo. 1 considers only the probability of the identifier (subtoken subsequence) substitution predicted by CodeBERT when generating the intermediate sets (full-permutation) and the merged set. To further facilitate the quality of the substitution, we filter \mathcal{S} with CodeBERT-MLM, selecting only identifiers with high perplexities (PPL). PPL is a widely-adopted metric in NLP to measure the quality of the generated examples. Given a sequence example w_1, \dots, w_l , where each w_i is a subtoken, we compute PPL with CodeBERT as below.

$$PPL(w_1, \dots, w_l) = \sqrt[l]{\prod_{i=1}^l \frac{1}{\text{prob}_{\text{CodeBERT}}(w_i | w_1, \dots, w_{i-1})}} \quad (11)$$

where $\text{prob}_{\text{CodeBERT}}(w_i | w_1, \dots, w_{i-1})$ is the probability of w_i predicted by CodeBERT given the prefix (w_1, \dots, w_{i-1}) . Note that we feed the subtoken sequence of the identifier only, rather than the whole code snippet, into CodeBERT. Because the identifier is often short (4-5 subtokens at most), the filtration by CodeBERT PPL is not the computational bottle-neck.

Mask. Unlike CodeBERT, whose (subtoken) vocabulary already includes <MASK>, most DL models do not have <MASK> in their vocabulary, which would cause problems for masked classification in CBA. However, there is often an unknown (<UNK>) in the vocabulary, due to the OOV issue of the DL models. Hence, we take advantage of <UNK> for our masked classification, by replacing the identifiers with <UNK>, which achieve the same effect as masking.

Simulated annealing. The probing phase in CBA is equivalent to greedy hill descending. During our initial experiments, we observe that this greedy search would cause early termination of CBA, because the algorithm gets stuck in some local optimal cases. *i.e.*, when the algorithm cannot decrease the predicted probability of the victim model, it terminates and returns none (see Line 22 in Algo. 1) To jump out of such local minimizations, we introduce simulated annealing⁶⁰ into CBA, forming CBA-SA. In the probing phase, CBA-SA selects the substitution with the lowest probability of the ground-truth class predicted by the victim model from the candidate substitutions, forming token sequence x' (where the identifier is already substituted). x' is accepted for the i -th iteration (*i.e.*, $x_i \leftarrow x'$) according to the acceptance rate α , formulated as:

$$\alpha = \begin{cases} e^{-\frac{C_y(x') - C_y(x_{i-1})}{T_{\text{iter}}}}, & \text{if } C_y(x') \geq C_y(x_{i-1}) \\ 1, & \text{otherwise} \end{cases} \quad (12)$$

where x_{i-1} comes from the $i-1$ -th iteration, T_{iter} is the temperature that decreases exponentially as the iteration grows, defined as $T_{\text{iter}} = T_0 \cdot \gamma^{\text{iter}}$, and $\gamma \in (0, 1)$ is the cooling factor. When $C_y(x') < C_y(x_{i-1})$, x' is always accepted; otherwise x' is very likely to be accepted in the beginning iterations, but rejected when the algorithm already run a couple of iterations. Simulated annealing enables CBA to make bold attempts in the very first iterations while keeping searching the descending directions.

5 | EVALUATION

We implement CBA as an extensible adversarial attack framework in Python with more than 5,000 lines of code based on the DL framework PyTorch (ver.1.4.0) and the transformers package⁷ (ver.3.3.0). With CBA, we performed a large-scale study to investigate the following research questions:

⁷<https://github.com/huggingface/transformers>

TABLE 2 Statistics of the subject tasks and datasets.

Dataset	Train #	Test #	Class #	Vocab #	LOC
OJ	~39K	~10K	104	~10K	~1.9M
OJClone	~1.3M	~5K	2	~10K	~1.9M

TABLE 3 Performance of the subject models.

Model	OJ(LM)	OJ(CLS)	OJClone	
	PPL	Acc(%)	Acc(%)	F1(%)
LSTM	–	95.3	87.4	87.3
CodeBERT-MLM	1.056	98.6	86.7	86.7
CodeBERT	–	98.8	87.2	87.2
GraphCodeBERT	–	97.6	84.4	85.1

RQ1: Attack effectiveness. Is CBA capable of performing black-box adversarial attacks effectively for source code processing? Can CodeBERT boost the searching process of adversarial example generation over the random MHM?

RQ2: Robustness of pre-trained models. Are the pre-trained model and their downstream models robust against adversarial attack? Whether and to what extent can the large pre-trained corpus and large parameter size help the pre-trained model to gain resilience against adversarial perturbations?

RQ3: Attack efficiency. Can CBA perform adversarial attack efficiently? Does MHM lack of efficiency indeed due to simple randomness, and can CBA outperform MHM by utilizing CodeBERT?

RQ4: Robustness enhancement through pre-training. Can pre-trained models gain resilience from designed pre-training tasks? To what extent the RTD pre-training task helps pre-trained model to enhance resilience against adversarial perturbations such as identifier substitution? Is it plausible to integrate adversarial training into pre-training?

RQ5: Quality of Adversarial Examples. Are adversarial examples generated by CBA with high quality? Compared with MHM, are the perturbations natural from the perspective of human programmers?

5.1 | Experimental Settings

Subject task and dataset. We select two source code classification tasks and datasets from CodeXGLUE benchmark²³ (i.e. functionality classification OJ^{1,2} and code clone detection OJClone^{3,2,4,5}), which are representative ones by DL. Table 2 summarizes the statistics of the subject datasets, which contain more than 1.9 million LOC in total. In particular, the original testset of OJClone in CodeXGLUE contains about 32K code pairs, but is highly unbalanced. Therefore, we downsampled 4.8K pairs forming the balanced testset. We follow the procedure in CodeXGLUE to pre-process OJ and OJClone for further analysis. In OJ, we employ classification accuracy (Acc) as the performance indicator of the DL models, and in OJClone, we employ F1-score as the indicator. These indicators are widely adopted in previous researches^{1,2,3,4,5}.

Subject model. We employ LSTM, CodeBERT and GraphCodeBERT as our subject (victim) models. LSTM is one of the most classic and representative sequential model adopted in the SE community. Although modern models do not directly apply the LSTM architecture, it is often utilized as the backbone to process sequence signals^{2,8,13}. For instance, the state-of-the-art Code2Seq¹³ employs LSTM to process the node paths in the AST. Therefore, to study the universality of the sequential models, we perform adversarial attack upon the LSTM architecture. As for CodeBERT²⁰ and GraphCodeBERT²¹, they are recently proposed pre-trained models for source code. As introduced in Sec. 3.3, The models consist of 12 layers of transformers. The “pre-training and fine-tuning” paradigm of CodeBERT and GraphCodeBERT has been proven very useful on many SE tasks, as presented in the recently proposed benchmark of CodeXGLUE²³.

We follow the previous work instruction to set up the training configurations for LSTM²⁵. Before fine-tuning in OJ and OJClone, CodeBERT and GraphCodeBERT are first fine-tuned in the manner of MLM with OJ, as we need to adapt the model to the distribution of C/C++ language. The fine-tuned CodeBERT-MLM for MLM is incorporated into CBA. In particular, we follow the open-sourced code provided in CodeXGLUE to fine-tune the CodeBERT models in OJClone. During fine-tuning, we perform binary classification, and during inference, CodeBERT makes predictions

TABLE 4 Adversarial attack against LSTM.

Adv.	OJ			OJClone		
	Succ(%)	Acc(%)	$\Delta(\%)^\dagger$	Succ(%)	F1(%)	$\Delta(\%)^\dagger$
None	–	95.3	–	–	87.3	–
MHM	29.1	65.4	32.4	30.7	54.6	37.5
CBA	47.0	48.9	48.7	30.0	55.6	36.3
CBA-SA	51.0	45.2	52.6	32.4	53.0	39.3

† In OJ, $\Delta = 1 - \frac{\text{Acc}}{\text{Acc}_{\text{None}}}$, while in OJClone, $\Delta = 1 - \frac{\text{F1}}{\text{F1}_{\text{None}}}$. High Δ suggests the effectiveness of the corresponding adversarial attack.

based on a selected threshold, which is selected in validation set – if the produced probability is greater than the threshold, the input code pair is classified as a clone; otherwise, they are not a clone pair. In our experiment, the threshold is 0.01. The models achieve competitive results on the subject datasets (Table 3).

Baseline. We adopt MHM as the baseline for comparative studies, which is already introduced in Sec. 3.2.

Evaluation metrics. We adopt attack success rate (Succ) as the major performance indicator of adversarial attack. Formally, it is computed as $\text{Succ} = \frac{|\mathcal{D}_{\text{correct}} \cap \hat{\mathcal{D}}_{\text{error}}|}{|\mathcal{D}_{\text{correct}}|}$, where $\|\cdot\|$ refers to the example number of a set, $\mathcal{D}_{\text{correct}}$ is the subset of the testset in which the examples are initially correctly classified by the victim model, and $\hat{\mathcal{D}}_{\text{error}}$ is another subset in which the examples are erroneously classified after adversarial attack. Perceptually, attack success rate suggests the powerfulness of the attack approach to mislead the victim model from originally correct classifications to erroneous behaviors. Besides, we also take the performance decreasing rate of the victim model before and after adversarial attack into consideration. This indicator intuitively shows the overall performance change of the victim after attack.

Experimental configuration. We leverage CBA to perform large-scale comparative experiments to answer the research questions. For **RQ1**, we attack LSTM with CBA and CBA-SA, along with the MHM baseline, in OJ and OJClone, to demonstrate the effectiveness of CBA. For **RQ2**, we attack the fine-tuned CodeBERT-MLM and GraphCodeBERT, to verify whether large model and large pre-training corpus is able to obtain resilience against adversarial attack, compared with traditional from-scratch trained models, such as LSTM. For **RQ3**, we analyze the attack processes from above, to demonstrate the efficiency of CBA. For **RQ4**, we attack CodeBERT and CodeBERT-MLM with CBA. By comparing them, we may verify whether the RTD pre-training task benefits the fine-tuned robustness against CBA. In addition, this RQ leads to an idea of “adversarial pre-training”⁶¹, where adversarial training is incorporated into the pre-training stage rather than fine-tuning. The fine-tuned model may inherit the resilience against attack, without additional adversarial training, which is efficient and friendly to the downstream task practitioners. For **RQ5**, we carry out human evaluation against adversarial examples generated by CBA and MHM. Specifically, we obtain examples from OJ which can be successfully attacked by both CBA and MHM against LSTM, and filter out those long sequence with the token length threshold of 80, leading to 38 example tuples. Each tuple consists of the original source code snippet, which can be correctly classified by the LSTM model, and the identifier substitutions generated by CBA and MHM, which can successfully mislead the LSTM model to erroneous prediction. Finally, we invite four PhD students as the independent volunteers, who all have at least three years of experience of C/C++ programming, to evaluate the example tuples from the views of naturalness and consistency.

We attack the victim models over the whole testsets. For OJClone, we carry out two trials upon a code pair, and each trial perturbs only one piece of code. We limit the iteration size to 20 and candidate size to 10. This setting is to better compare the efficiency of CBA with the MHM baseline. Please refer to Sec. 5.4 for more comprehensive discussions. In CBA, we set the vulnerable size to 5 at most. As for the simulated annealing process, we set the initial temperature T_0 to 1 and the cooling factor γ to 0.8. All the experiments were run on a server of Ubuntu 18.04 system with 32-core 2.50GHz Xeon Platinum CPU, 128 GB RAM and 4 NVIDIA Tesla V100 16GB GPUs.

5.2 | RQ1: Attack Effectiveness

Effectiveness. Table 4 shows the performance of LSTM before and after adversarial perturbations by CBA and CBA-SA, along with the random baseline MHM. The attack success rate (“Succ(%)” in Table 4) is computed only over the originally correctly predicted examples. On average, CBA and CBA-SA reduce the performance of LSTM by 42.5% and 46.0%, outperforming the MHM baseline (35.0%). As for attack success rate, CBA and CBA-SA successfully mislead LSTM at rates of 38.5% and 41.7%, also exceeding MHM (29.9%). In particular, in OJ, CBA-SA is capable of fooling the LSTM model at a success rate of more than 50%, even when the iteration size and the candidate size are rather small. Furthermore, in OJ, we

TABLE 5 Examples of adversarial perturbations against LSTM in OJ.

Source Code	ID Substitution
<pre> int main(){ int n; cin >> n; int a[n]; for(int i=0; i<n; i++){ cin >> a[i]; } if(n==1) cout << a[0] << endl; else { cout << a[n-1]; for(int i=n-2; i>=0; i--){ cout << " " << a[i]; } } return 0; } </pre>	MHM (Class 46⇒91)
	i → shijianbiao
	CBA (Class 46⇒101)
	n → t
	a → as
	CBA-SA (Class 46⇒101)
	n → t
	a → as

even test MHM with a large candidate size (50), denoted as MHM(50), which is much larger than the configurations in Sec. 5.1 (candidate size = 10). The attack success rate of MHM(50) is 43.3%, and it reduces the accuracy of LSTM to 53.2%, which is still outperformed by CBA and CBA-SA. Therefore, we can see that CBA demonstrates its effectiveness in OJ and OJClone against LSTM models.

Adversarial example. Table 5 presents a typical set of adversarial perturbations produced by MHM and CBA. The original code outputs a sequence of integers in reversed order. MHM substitutes “i” with “shijianbiao” (Chinese pinyin for “schedule”), misleading LSTM from 46 (correct) to 91. Such perturbation is random and easy to be distinguished, as no programmers would name a loop variable to “schedule”, and therefore such perturbations are likely to have rather low quality. In comparison, CBA substitutes “n” and “a” with “t” and “as” respectively, resulting in LSTM’s erroneous prediction to class 101. Such identifier perturbations may have higher quality, because “t” for input size and “as” for array are much more common. In addition, CBA-SA can cover any perturbation produced by CBA, therefore they produce identical examples in Table 5. More cases of MHM and CBA against LSTM are presented in Sec A. Please refer to the appendix. Besides the cases, we also carry out human evaluation in RQ5 to demonstrate the perturbation quality of CBA. Please refer to Sec. 5.6 for more discussions.

Answer to RQ1: CBA is capable of performing effective adversarial attack against the popular LSTM model in OJ. CBA is much more effective than the MHM method.

5.3 | RQ2: Robustness of Pre-Trained Models

Effectiveness. Table 6 summarizes the performance of CodeBERT-MLM in OJ and OJClone before and after adversarial perturbations. Similar conclusions to Table 4 can be drawn. The success rates of CBA and CBA-SA on average are 15.6% and 17.6%, outperforming MHM baseline (14.3%), while the performance of CodeBERT-MLM after CBA and CBA-SA decreases 15.4% and 17.5%, respectively, also outperforming MHM (14.7%).

Robustness of CodeBERT-MLM. Comparing Table 4 and 6, we formulate the robustness gaining of CodeBERT-MLM (Table 7). On average, the attack success rates of all three approaches decrease 71.6% and 37.1% in OJ and OJClone, respectively, when the victim model changes from LSTM to CodeBERT-MLM. Such results indicate that pre-training with MLM on a large corpus may help DL models to gain resilience against identifier substitution attack.

Adversarial example. Please refer to Sec. A in the appendix for cases of perturbations by MHM and CBA against CodeBERT-MLM. Similar findings as Sec. 5.2 can be found. Due to randomness, MHM substitutes identifiers to completely unrelated ones (e.g., “c” → “EVEN”), neglecting the original textual and contextual information. However, CBA performs much more reasonable substitutions by considering the context. This is because

TABLE 6 Adversarial attack against CodeBERT-MLM.

Adv.	OJ			OJClone		
Atk.	Succ(%)	Acc(%)	$\Delta(\%)^\dagger$	Succ(%)	F1(%)	$\Delta(\%)^\dagger$
None	-	98.6	-	-	86.7	-
MHM	8.1	89.5	9.2	<u>20.4</u>	<u>69.2</u>	<u>20.2</u>
CBA	13.0	84.7	14.1	18.1	72.2	16.7
CBA-SA	<u>15.1</u>	<u>82.6</u>	<u>16.2</u>	20.1	70.5	18.7

† In OJ, $\Delta = 1 - \frac{Acc}{Acc_{None}}$, while in OJClone, $\Delta = 1 - \frac{F1}{F1_{None}}$. High Δ suggests the effectiveness of the corresponding adversarial attack.

TABLE 7 Adversarial robustness gaining of CodeBERT-MLM from pre-training.

Adv.	OJ Atk. Succ (%)			OJClone Atk. Succ(%)		
Atk.	LSTM	CB †	$\Delta(\%)^\diamond$	LSTM	CB †	$\Delta(\%)^\diamond$
MHM	29.1	8.1	+72.2	30.7	20.4	+33.6
CBA	47.0	13.0	+72.3	30.0	18.1	+39.7
CBA-SA	51.0	15.1	+70.4	32.4	20.1	+38.0

† CodeBERT-MLM, pre-trained through MLM only.

$^\diamond$ $\Delta = 1 - \frac{CB}{LSTM}$. Positive Δ indicates resilience against adversarial perturbations of CodeBERT-MLM.

TABLE 8 Adversarial errors of CodeBERT-MLM in OJClone

Err. Pred.	MHM	CBA	CBA-SA
Not clone \rightarrow clone	654	734	815
Clone \rightarrow not clone	1,045	775	861

the CodeBERT-MLM language model adapts to the distribution of C/C++ programming language, and generates more natural and consistent candidates.

Threshold bias in OJClone. The threshold of CodeBERT in OJClone is likely to cause the rather low but competitive performance of CBA compared with MHM (Table 6). As introduced in Sec. 5.1, CodeBERT on OJClone plays the role of a binary classifier, which outputs a clone probability and predicts according to a threshold (automatically selected on the validation set). Intuitively, the threshold is supposed to be near 0.5 for a binary classification, but in our OJClone experiment, the threshold of CodeBERT is heavily biased (0.01).⁸

The threshold bias indicates that CodeBERT fine-tuned upon OJClone is not confident about its clone prediction, because the clone probability is heavily offset to 0 (leading to the threshold of 0.01). Therefore, misleading CodeBERT from the clone prediction to non-clone (C2N) is much easier than from non-clone to clone (N2C). After this analysis, we study the case numbers of C2N and N2C of MHM, CBA, and CBA-SA (Table 8). Table 8 suggests that MHM takes the advantage of the threshold bias by performing the easier C2N attack almost twice than the N2C flipping. However, CBA does not show any obvious preference to the C2N attack, as the number of C2N and N2C are comparable.

Robustness of GraphCodeBERT. It is possible that CBA is specialized to attack CodeBERT-MLM only, since in the previous experiments, CBA uses CodeBERT-MLM’s spear to attack its own shield. Therefore, we further apply CBA to attack another pre-trained model, i.e., GraphCodeBERT. The results are shown in Table 9. The success attack rates of CBA against GraphCodeBERT are 7.2% and 24.4% on OJ and OJClone respectively, and

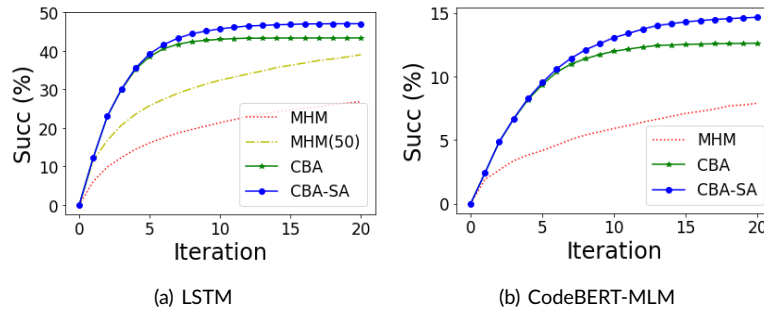
⁸We strictly follow the guidance of CodeXGLUE. The authors of CodeXGLUE²³ have also confirmed this phenomenon. Therefore, such bias is not likely to be caused by our experimental procedure.

TABLE 9 Adversarial robustness gaining of GraphCodeBERT from pre-training.

Adv. Atk.	OJ Atk. Succ (%)			OJClone Atk. Succ(%)		
	LSTM	GCB [†]	$\Delta(\%)^\diamond$	LSTM	GCB [†]	$\Delta(\%)^\diamond$
CBA	47.0	7.2	+84.7	30.0	24.4	+18.7
CBA-SA	51.0	8.1	+84.1	32.4	25.5	+21.3

[†] GraphCodeBERT, pre-trained through MLM, data flow node alignment, and data flow edge prediction tasks. The last two tasks force GraphCodeBERT to learn the data flow information.

[◊] $\Delta = 1 - \frac{GCB}{LSTM}$. Positive Δ indicates resilience against adversarial perturbations of GraphCodeBERT.

**FIGURE 3** Attack success rate of adversarial attack algorithms against LSTM and CodeBERT-MLM in OJ. The higher curve suggests the more effective and efficient black-box attack algorithm.

the simulated annealing trick in CBA-SA further increase them to 8.1% and 25.8%. Therefore, CBA is still capable to attack pre-trained models other than CodeBERT-MLM itself.

Additionally, we compare the performance of LSTM and GraphCodeBERT in order to demonstrate from another side that large model and large corpus pre-training may enhance the robustness. The comparison is presented in Table 9. When the victim model changes from LSTM to GraphCodeBERT, the attack success rate on average decreases 84.4% and 20.0% upon OJ and OJClone separately. Such results also suggest that the large model and the large corpus pre-training may provide resilience against attack in the downstream task.

Answer to RQ2: The fine-tuned large pre-trained models are more robust than traditional end-to-end LSTM in OJ and OJClone against CBA and MHM. However, there are still non-robust drawbacks within CodeBERT-MLM, such as the threshold design for OJClone.

5.4 | RQ3: Attack Efficiency

Efficiency. We plot the attack success rate curves against iteration of all adversarial attack algorithms against LSTM and CodeBERT-MLM in OJ (Fig. 3). The computational bulk in CBA along with the MHM baseline is to probe the victim model. Therefore, the number of calls to the victim model before a successful attack indicates the efficiency of the attack approach. As we set the candidate size of both CBA and MHM to be identical, the success-iteration curve also indicates the efficiency – the higher of the success-iteration curve, the more efficient the attack algorithm. CBA and CBA-SA saturate after about 15 iterations, indicating that configurations in Sec. 5.1 is reasonable. Curves of MHM are much lower and rise much slower than CBA, indicating that CBA is much more efficient than MHM method. In addition, we do not employ the actual running time as the major metric, because it is highly dependent on the condition of the machine (such as workload) at the time.

Randomness in MHM. To further demonstrate the drawback of simple random strategy in MHM, we dig into the sampling process, discovering that in the 20 iterations on OJ, MHM in most cases only produces 1 or 2 accepted substitutions, while other proposals are all rejected by M-H. Therefore, MHM wastes most of its time upon useless rejected proposals. One major reason is that these proposals are randomly generated,

TABLE 10 Attack against CodeBERT pre-trained with MLM and RTD.

Adv. Atk.	OJ Atk. Succ (%)			OJClone Atk. Succ(%)		
	MLM [†]	+RTD [◊]	$\Delta(\%)^*$	MLM [†]	+RTD [◊]	$\Delta(\%)^*$
CBA	13.0	12.2	+6.2	18.1	15.6	+13.8
CBA-SA	15.1	14.3	+5.3	20.1	16.9	+15.9

[†] CodeBERT-MLM, pre-trained through MLM only.

[◊] CodeBERT, pre-trained through MLM and RTD, where MLM is adjunct to RTD.

^{*} $\Delta = 1 - \frac{RTD}{MLM}$. Positive Δ suggests that RTD may help CodeBERT gain resilience against adversarial attack.

TABLE 11 Attack against CodeBERT-MLM and GraphCodeBERT.

Adv. Atk.	OJ Atk. Succ (%)			OJClone Atk. Succ(%)		
	MLM [†]	+DF [◊]	$\Delta(\%)^*$	MLM [†]	+DF [◊]	$\Delta(\%)^*$
CBA	13.0	7.2	+44.6	18.1	24.4	-34.8
CBA-SA	15.1	8.1	+46.4	20.1	25.5	-26.9

[†] CodeBERT-MLM, pre-trained through MLM only.

[◊] GraphCodeBERT, pre-trained through MLM, node alignment and edge prediction (“DF” refers to “data flow”). The last two tasks make GraphCodeBERT to capture the data flow information.

^{*} $\Delta = 1 - \frac{DF}{MLM}$. Positive Δ suggests that data flow may help GraphCodeBERT gain resilience against adversarial attack.

which may not well fit the stationary (target) distribution. On the other hand, equipped with better searching guidance by CodeBERT, CBA keeps searching for plausible adversarial perturbations with much higher efficiency.

Simulated annealing. CBA-SA adapts simulated annealing to avoid getting stuck on local optima. As shown in Fig. 3, when CBA saturates, the curves of CBA-SA still increases. This is because simulated annealing enables CBA-SA to jump out of local cases and to continue the searching process. Eventually, CBA-SA achieves a higher attack success rate than CBA (as in Table 4 and 6).

Answer to RQ3: The proposed CBA is more efficient than the random MHM baseline to attack LSTM and CodeBERT-MLM. The incorporation of simulated annealing further improves the performance of CBA-SA without the loss of efficiency.

5.5 | RQ4: Robustness Enhancement

Effectiveness. Table 10 shows the attack success rate of CBA against CodeBERT-MLM and CodeBERT. Against CodeBERT, CBA-SA is more effective than CBA as expected. On average, CBA and CBA-SA produce success rates of 13.9% and 15.6%, respectively.

Robustness enhancement by RTD. In OJ, CodeBERT reduces the attack success rate of CBA and CBA-SA on average by 5.8% than CodeBERT-MLM, while in OJClone, 14.9%. The result indicates that CodeBERT pre-trained with MLM and RTD may be more adversarially robust than CodeBERT-MLM pre-trained only with MLM, as CodeBERT is harder to be attacked. This result is also explainable – RTD requires CodeBERT to distinguish those replaced tokens, therefore, the pre-trained CodeBERT naturally has the capability to detect the substitutions, and the fine-tuned model inherits the resilient to identifier substitution attack. Furthermore, the results indicates the feasibility of adversarial pre-training for source code processing.

Robustness enhancement by data flow. Compared with CodeBERT-MLM, GraphCodeBERT additionally introduces two per-training tasks based on the data flow. We compare CodeBERT-MLM and GraphCodeBERT to see whether data flow can improve the robustness of the fine-tuned model. The comparison results are shown in Table 11, which is quite subtle. In OJ, GraphCodeBERT reduces the attack success rate of CBA and CBA-SA

TABLE 12 Human evaluation of adversarial examples generated by MHM and CBA.

Volunteer (Pseudonym)	Naturalness (-2~2)		Consistency (-2~2)		MHM (-2) vs. CBA (2)	Comment
	MHM	CBA	MHM	CBA		
Jolyne	-1.58	1.42	-1.26	1.53	1.63	"The substitutions by MHM are complex, redundant and incoherent, while CBA is more natural and brief."
Magent	-1.79	0	-1.63	0.26	1.11	"MHM renames IDs like a beginner, while CBA produces more proper names for general OI programs programs."
Ciocolata	0.47	0.37	-0.58	-0.05	0.26	"MHM prefers bold or even aggressive substitutions, while CBA performs normal or even mediocre ones."
Q	-0.79	1.58	-1.11	1.42	1.74	"Compared to CBA, IDs renamed by MHM are too complicated, and do not conform the usual naming habits."
Average	-0.92	0.84	-0.93	0.79	1.19	-

on average by 45.5%, which is even significant than RTD in CodeBERT. However, in OJClone, the situation is reversed, as the average success rate against GraphCodeBERT increases 30.9% compared with CodeBERT. The results suggest the challenge of universal adversarial pre-training, where the pre-training task needs to be designed carefully.

Answer to RQ4: The pre-training RTD task is capable to help the model to gain adversarial resilience against identifier substitution adversarial attack. It suggests that adversarial pre-training is plausible. However, results of GraphCodeBERT also indicate the challenge of the design of the adversarial pre-training tasks.

5.6 | RQ5: Quality of Adversarial Examples

Questionnaire. The questionnaire for each volunteer consists of 20 sets of rating questions, each corresponding to an example tuple (original code snippet + MHM perturbations + CBA perturbations). We guarantee that each tuple is at least evaluated by two different volunteers. There are five rating questions for each example tuple. ❶ Are the perturbations generated by MHM natural to the volunteer? *I.e.*, Will the volunteer name the the same or similar identifiers as the substitutions by MHM? ❷ Similarly, are the perturbations produced by CBA natural? ❸ Are the identifiers substituted by MHM consistent with the context? In other words, are the textual meaning of the substituted identifiers appropriate in the context of this very snippet? ❹ Are the identifiers renamed by CBA consistent with the context? ❺ Does CBA generate perturbations with higher quality than MHM? The rating score ranges from -2 (not at all) to 2 (yes for sure), and 0 refers to neutral or "cannot decide". After completing the questionnaire, we communicate with the volunteers about how their feeling about the perturbations generated by CBA along with MHM.

Human evaluation. Table 12 lists the averaged rating scores from each volunteer (with pseudonym) and the overall results. Combining the opinions from the four volunteers, MHM is not satisfactory from the perspective of naturalness and consistency, as the rating scores are negative (about -0.9). However, the scores of CBA are both around 0.8, suggesting CBA produces much more natural and consistent perturbations than MHM. Moreover, all volunteers consider CBA with higher quality compared with MHM, as the positive rating scores of the fifth question indicate.

Besides the rating scores, Table 12 also presents the comments from the volunteers. The words sum up the volunteers' feeling about CBA and MHM. In general, MHM produces complicated and bold substitutions, while identifiers generated by CBA are more in line with the habits of human programmers and the distribution of programs in OJ. This is quite understandable, since MHM completely relies randomness without considering the context, while CBA leverages CodeBERT-MLM to generate substitutions according to the learned language distribution.

Answer to RQ5: Compared with MHM, from the perspective of human programmers, CBA generates more natural consistent adversarial examples. The identifier substitutions produced by CBA are with rather high-quality.

6 | DISCUSSION

In this section, we discuss the extensibility of CBA, *e.g.* languages other than C/C++ (Sec. 6.1) and perturbations other than identifier substitution (Sec. 6.2), and threats to validity (Sec. 6.5).

TABLE 13 Attack against CodeBERT for code defect prediction in Java.

Victim	None	CBA		CBA-SA	
Mode	Acc(%)	Succ(%)	Acc(%)	Succ(%)	Acc(%)
CodeBERT	93.1	17.8	75.1	19.7	73.3

TABLE 14 Attack success rate (%) with different types of perturbations.

Task	Perturb.	Adv. Atk.	LSTM	CB-MLM	CB
OJ	WS	CBA	24.0	3.8	4.5
		CBA-SA	23.6	4.0	4.4
	CMP	CBA	9.9	1.3	1.4
		CBA-SA	10.2	1.4	1.5
OJClone	WS	CBA	21.2	22.3	25.7
		CBA-SA	21.3	22.1	25.9

6.1 | CBA for Other Languages

For more general application scope extension of CBA, we do not limit it to only C/C++ language. Instead, we process the source code with tree-sitter⁹, which is an open-sourced multilingual parsing tool. Tree-sitter allows CBA to switch to another programming language quickly, as we only need to switch to another parsing-core for the corresponding language.

We also conduct a demonstrative experiment upon a self-constructed dataset based on Java-small⁶² for code defect prediction. We substitute comparison operator in code to create defects (e.g. " $<$ " \Rightarrow " $<=$ "), forming a binary classification task. CBA and CBA-SA are capable of carrying out adversarial attacks against CodeBERT in Java language (Table 13). Therefore, our proposed CBA is not limited to C/C++, but can be applied to other languages.

6.2 | CBA for Other Types of Perturbations

CBA can also be further extended to other types of semantic equivalent perturbations, such as white space insertion and comparison operand swapping, etc. White space insertion (WS) is to insert spaces (" "), tabs ("\t"), and newlines ("\n") into code. During our experiment, we find a possible flaw of BPE, as white spaces create empty subtokens as "". E.g., "int a;" after BPE would be {"int", "","a", ";"}, with two empty subtokens. Multiple empty subtokens may cause difficulties for the victim model to understand the code properly. We adapt masked classification for WS by inserting masks into the code, searching for vulnerabilities. Then, we perform white space substitutions upon the vulnerable positions, where the candidates consist of only white spaces. Comparison operand swapping (CMP) is to swap the operands of a comparison statement. E.g., " $a>b$ " after swapping would be " $b<a$ ". As for CMP, similarly, we mask the comparison expressions to perform masked classification, looking for vulnerable comparisons. Then we directly swap the operands of the vulnerable expressions, forming the candidate set.

We conduct a set of illustrative experiments with CBA equipped with the aforementioned perturbations (Table 14). In OJ, the CodeBERT models are capable of resisting the perturbations, as the success rates are lower than 5%, while in OJClone, all three models failed with attack success rates over 20%. In addition, CodeBERT pre-trained with RTD can not enhance the robustness against these two types of perturbations.

We would like to further discuss the adversarial perturbation for source code a bit more. Existing perturbations are usually fine-grained rule-based transformations, such as identifier renaming^{25,46,50,49}, dead-code insertion or deletion^{46,50}, and equivalent structure transformation (for \Leftrightarrow while)⁴⁷. There are two major reasons why we do not consider in a large scale, e.g., rewrite. ❶ The adversarial perturbation cannot violate the grammar. Existing techniques such as back-translation have the capability to rewrite the code, but they can hardly guarantee 100% correctness. The high-level perturbation is still very challenging, and more in-depth progress of the field is required. ❷ Apart from the correctness constraint, we also would like the perturbation to be imperceptible to human programmers. Therefore, the granularity of the perturbation is not supposed to be too large.

⁹<https://github.com/tree-sitter/tree-sitter>

6.3 | Adversarial Pre-Training

Adversarial pre-training incorporates adversarial perturbation into the pre-training process, hoping to gain resilience and pass such resilience to the fine-tuned model against adversarial attack. Compared with traditional adversarial training, there are two major advantages of adversarial pre-training. ❶ Adversarial pre-training integrates adversary during the pre-training stage, avoiding burdening the downstream task practitioners. Ideally, the downstream practitioner needs only focuses on the downstream task itself, and the fine-tuned model (without additional adversarial training) is capable to resist adversarial attack. It can be friendly and efficient for the downstream task practitioners. ❷ During the pre-training stage, the corpus is extremely large. Therefore, multiple perturbations or even a universal perturbation can be involved during adversarial pre-training. It is rather hard and resource-consuming to achieve so in the downstream tasks. So far, a plausible research direction is through contrastive learning⁶³, where the representation of the positive pair (similar or even the same samples) gets closer, while the distance of the negative one is increased. Please refer to the survey of adversarial pre-training for more comprehensive introduction⁶¹.

In this paper, we attempt to verify the idea of adversarial pre-training in the context of source code processing. CodeBERT provides a perfect prototype experiment. The MLM model (CodeBERT-MLM) provides token substitution as perturbation, and the RTD model (CodeBERT) tries to identify such substitution, creating a natural scenario of adversarial pre-training. Our experimental results indicates that without additional adversarial training, the fine-tuned CodeBERT inherits the resilience against identifier renaming from the RTD pre-training task. Adversarial pre-training is a new research direction, and we will try to design a universal perturbation for adversarial pre-training in our future work.

6.4 | Exact Black-Box Attack

In this paper, as a rather early investigation about adversary of code model, we make a relatively broad categorization of the scenario – white-box (the internal states of the DL model, such as parameters and gradients, are available) and black-box (the internal states are not accessible). But in other field, the black-box scenario is further divided into gray-box and exact black-box⁶⁴ – the gray-box setting allows access to the prediction probability of the victim model, while the exact black-box setting only makes the discrete prediction available. At a rather early stage, following the previous MHM, we do not constrain victim model output to be discrete classification. As our future work, we may try to tackle the more challenging exact black-box attack against source code processing models.

It is possible that we may leverage CBA to produce adversarial perturbation in the exact black-box scenario, but it is almost predictably not ideally effective or efficient. As aforementioned, we may regard the exact black-box scenario as a special case of gray-box. The discrete classification can be considered as a one-hot probability distribution, i.e., the probability of the predicted class is 1 and others 0. Therefore, it is plausible to migrate CBA to exact black-box attack, but with great challenges. ❶ During vulnerability detection, CBA still employs masked classification to compute the vulnerability score. But without accessing the prediction probability, the vulnerability manifests only when the predicted class changes, resulting a very small or even empty subset of the vulnerabilities. ❷ In the candidate generation and probing phase, the hill-climbing search will almost fail, as the searching surface becomes a platform (probability 1) with only a few holes (probability 0). The probability changes only when it falls like a staircase from 1 to 0. Thence, the exact black-box setting is very challenging to CBA, and we will try to tackle this new problem in the future.

6.5 | Threat to Validity

Threat to external validity. The quality of our subject tasks and the representative of the victim models are major threats to external validity. ❶ The subject code classification (OJ) and clone detection (OJClone) tasks have been studied in previous work of adversarial attack against source code models²⁵. The two datasets are also included in the widely accepted CodeXGLUE benchmark²³. Besides OJ and OJClone (both C/C++ language), we also extend to a manually crafted defect prediction task in Java to demonstrate the capability of CBA. However, further studies upon other tasks and programming languages are still required to further generalize our findings. ❷ The studied subject (victim) LSTM and CodeBERT models are popular architectures in the field of source code processing. LSTM is one of the most classic and popular architecture, and it is the backbone of many state-of-the-art models^{2,8,13}. CodeBERT, employing the transformer architecture, is a widely adopted and studied pre-trained model for source code^{20,21,23}. From the perspective of architecture, our victim selection covers the recurrent sequential model (LSTM) and the transformer (CodeBERT); from the perspective of learning, the victim selection covers the end-to-end “from scratch” paradigm (LSTM) and the “pre-training and fine-tuning” paradigm (CodeBERT). However, to further facilitate generalizability of our conclusion, other architecture need to be evaluated.

Threat to internal validity. The hyper-parameter selection of both CBA and MHM is a threat to internal validity. Besides, the type of adversarial perturbation can be a threat too. ❶ The iteration budget is set to 20, which may affect the convergence of the adversarial attack approaches. As shown in Fig. 3, CBA converges in the given iteration budget. However, due to the limitation of computational resource, the vulnerable size, i.e., $n_{vul} = 5$, and the candidate set size, i.e., $n_{cand} = 10$, are rather small. To find the best hyper-parameter combination and to achieve the best performance of CBA, a grid searching over n_{vul} , n_{inter} , and n_{cand} is required. ❷ For a fair comparison with CBA, we limit the candidate size of

MHM to 10, which may cause the MHM baseline to perform poorly. To counter this threat, we carry an experiment, where the candidate size of MHM is even enlarged to 50 (Fig. 3(a)). The comparison justifies the capability of CBA. ④ CBA performs identifier renaming perturbations upon source code, which may not be general. Other types of perturbations are possible to reach different conclusions. Therefore, we extend CBA to other types of code perturbations in Sec. 6.2. Further extension of CBA is required to examine the general robustness of the models.

Threat to construct validity. The evaluation metrics of both the victim models and the adversarial attack approaches can be threats to construct validity. ① Accuracy and F1-score are employed as the performance indicators for OJ and OJClone respectively. They are classic evaluation measurements and are adopted in the previous work^{2,3,25}. ② We employ attack success rate and performance decreasing of the victim model as the performance indicators of the adversarial attack approaches. Both indicators have been adopted in the previous work of adversarial attack against source code model – attack success rate from MHM²⁵ and performance decreasing from DAMP⁴⁶.

7 | CONCLUSION

This paper proposes CBA, a black-box identifier-substitution adversarial attack approach equipped with a large pre-trained CodeBERT model, for the context of source code processing. Our in-depth evaluation confirms the effectiveness and efficiency of CBA. Inspired by the idea of “turning CodeBERT against CodeBERT”, we further conduct adversarial attacks against CodeBERT models, demonstrating the robustness gaining of CodeBERT through pre-training.

References

1. Mou L, Li G, Zhang L, Wang T, Jin Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In: Schuurmans D, Wellman MP., eds. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA* AAAI Press; 2016: 1287–1293.
2. Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X. A novel neural source code representation based on abstract syntax tree. In: Atlee JM, Bultan T, Whittle J., eds. *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019* IEEE / ACM; 2019: 783–794
3. Wei H, Li M. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In: Sierra C., ed. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017* ijcai.org; 2017: 3034–3040
4. Yu H, Lam W, Chen L, Li G, Xie T, Wang Q. Neural detection of semantic code clones via tree-based convolution. In: Guéhéneuc Y, Khomh F, Sarro F., eds. *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019* IEEE / ACM; 2019: 70–80
5. Wang W, Li G, Ma B, Xia X, Jin Z. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In: Kontogiannis K, Khomh F, Chatzigeorgiou A, Fokaefs M, Zhou M., eds. *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020* IEEE; 2020: 261–271
6. Allamanis M, Peng H, Sutton CA. A Convolutional Attention Network for Extreme Summarization of Source Code. In: Balcan M, Weinberger KQ., eds. *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 48 of *JMLR Workshop and Conference Proceedings*. JMLR.org; 2016: 2091–2100.
7. Alon U, Zilberstein M, Levy O, Yahav E. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 2019; 3(POPL): 40:1–40:29. doi: 10.1145/3290353
8. Li J, Wang Y, King I, Lyu MR. Code Completion with Neural Attention and Pointer Networks. *CoRR* 2017; abs/1711.09573.
9. Liu F, Li G, Wei B, Xia X, Fu Z, Jin Z. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. In: ACM; 2020: 37–47
10. Liu F, Li G, Zhao Y, Jin Z. Multi-task Learning based Pre-trained Language Model for Code Completion. In: IEEE; 2020: 473–485.

11. Hu X, Li G, Xia X, Lo D, Jin Z. Deep code comment generation. In: Khomh F, Roy CK, Siegmund J., eds. *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018ACM*; 2018: 200–210
12. Hu X, Li G, Xia X, Lo D, Lu S, Jin Z. Summarizing Source Code with Transferred API Knowledge. In: Lang J., ed. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Swedenijcai.org*; 2018: 2269–2275
13. Alon U, Brody S, Levy O, Yahav E. code2seq: Generating Sequences from Structured Representations of Code. In: OpenReview.net; 2019.
14. Svyatkovskiy A, Deng SK, Fu S, Sundaresan N. IntelliCode compose: code generation using transformer. In: Devanbu P, Cohen MB, Zimmermann T., eds. *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020ACM*; 2020: 1433–1443
15. Devlin J, Chang M, Lee K, Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Burstein J, Doran C, Solorio T., eds. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)Association for Computational Linguistics*; 2019: 4171–4186
16. Liu Y, Ott M, Goyal N, et al. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* 2019.
17. Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training.; 2018.
18. Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I. Language models are unsupervised multitask learners. *OpenAI blog* 2019; 1(8): 9.
19. Brown TB, Mann B, Ryder N, et al. Language Models are Few-Shot Learners. In: Larochelle H, Ranzato M, Hadsell R, Balcan M, Lin H., eds. *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*; 2020.
20. Feng Z, Guo D, Tang D, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In: Association for Computational Linguistics; 2020; Online: 1536–1547
21. Guo D, Ren S, Lu S, et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. *CoRR* 2020; abs/2009.08366.
22. Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* 2019.
23. Lu S, Guo D, Ren S, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* 2021.
24. Zhou Y, Liu S, Siow JK, Du X, Liu Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In: Wallach HM, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox EB, Garnett R., eds. *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*; 2019: 10197–10207.
25. Zhang H, Li Z, Li G, Ma L, Liu Y, Jin Z. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In: ; 2020.
26. Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E. Equation of state calculations by fast computing machines. *The journal of chemical physics* 1953; 21(6): 1087–1092.
27. HASTINGS W. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 1970; 57(1): 97–109.
28. Chib S, Greenberg E. Understanding the metropolis-hastings algorithm. *The american statistician* 1995; 49(4): 327–335.
29. Allamanis M, Barr ET, Devanbu PT, Sutton C. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 2018; 51(4): 81:1–81:37. doi: 10.1145/3212695
30. Wang S, Chen Z, Yu X, et al. Heterogeneous Graph Matching Networks for Unknown Malware Detection. In: Kraus S., ed. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019ijcai.org*; 2019: 3762–3770
31. Pradel M, Sen K. DeepBugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.* 2018; 2(OOPSLA): 147:1–147:25. doi: 10.1145/3276517

32. Hellendoorn VJ, Bird C, Barr ET, Allamanis M. Deep learning type inference. In: Leavens GT, Garcia A, Pasareanu CS., eds. *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*ACM; 2018: 152–162
33. Vaswani A, Shazeer N, Parmar N, et al. Attention is All You Need. In: NIPS'17. Curran Associates Inc.; 2017; Red Hook, NY, USA: 6000–6010.
34. Peters ME, Neumann M, Iyyer M, et al. Deep Contextualized Word Representations. In: Walker MA, Ji H, Stent A., eds. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*Association for Computational Linguistics; 2018: 2227–2237
35. Fedus W, Zoph B, Shazeer N. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *arXiv preprint arXiv:2101.03961* 2021.
36. Karampatsis R, Sutton C. SCELMO: Source Code Embeddings from Language Models. *CoRR* 2020; abs/2004.13214.
37. Szegedy C, Zaremba W, Sutskever I, et al. Intriguing properties of neural networks. In: Bengio Y, LeCun Y., eds. *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*; 2014.
38. Goodfellow IJ, Shlens J, Szegedy C. Explaining and Harnessing Adversarial Examples. In: Bengio Y, LeCun Y., eds. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*; 2015.
39. Kurakin A, Goodfellow IJ, Bengio S. Adversarial examples in the physical world. In: OpenReview.net; 2017.
40. Papernot N, McDaniel PD, Jha S, Fredrikson M, Celik ZB, Swami A. The Limitations of Deep Learning in Adversarial Settings. In: IEEE; 2016: 372–387
41. Alzantot M, Sharma Y, Elgohary A, Ho B, Srivastava MB, Chang K. Generating Natural Language Adversarial Examples. In: Riloff E, Chiang D, Hockenmaier J, Tsujii J., eds. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*Association for Computational Linguistics; 2018: 2890–2896
42. Ebrahimi J, Rao A, Lowd D, Dou D. HotFlip: White-Box Adversarial Examples for Text Classification. In: Gurevych I, Miyao Y., eds. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 2: Short Papers*Association for Computational Linguistics; 2018: 31–36
43. Zhang H, Zhou H, Miao N, Li L. Generating Fluent Adversarial Examples for Natural Languages. In: Korhonen A, Traum DR, Màrquez L., eds. *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*Association for Computational Linguistics; 2019: 5564–5569
44. Li L, Ma R, Guo Q, Xue X, Qiu X. BERT-ATTACK: Adversarial Attack Against BERT Using BERT. In: Webber B, Cohn T, He Y, Liu Y., eds. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*Association for Computational Linguistics; 2020: 6193–6202
45. Zang Y, Qi F, Yang C, et al. Word-level Textual Adversarial Attacking as Combinatorial Optimization. In: Jurafsky D, Chai J, Schluter N, Tetreault JR., eds. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*Association for Computational Linguistics; 2020: 6066–6080
46. Yefet N, Alon U, Yahav E. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 2020; 4(OOPSLA): 162:1–162:30. doi: 10.1145/3428230
47. Applis L, Panichella A, Deursen vA. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In: IEEE; 2021: 1377–1381
48. Srikant S, Liu S, Mitrovskaya T, et al. Generating Adversarial Computer Programs using Optimized Obfuscations. In: OpenReview.net; 2021.
49. Yang Z, Shi J, He J, Lo D. Natural Attack for Pre-trained Models of Code. In: ACM; 2022: 1482–1493
50. Zhang H, Fu Z, Li G, et al. Towards Robustness of Deep Program Processing Models - Detection, Estimation, and Enhancement. *ACM Trans. Softw. Eng. Methodol.* 2022; 31(3): 50:1–50:40. doi: 10.1145/3511887

51. Zeng Z, Tan H, Zhang H, Li J, Zhang Y, Zhang L. An extensive study on pre-trained models for program understanding and generation. In: Ryu S, Smaragdakis Y., eds. *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*ACM; 2022: 39–51
52. Carlini N, Wagner DA. Towards Evaluating the Robustness of Neural Networks. In: IEEE Computer Society; 2017: 39–57
53. Bubeck S, Lee YT, Price E, Razenshteyn IP. Adversarial examples from computational constraints. In: Chaudhuri K, Salakhutdinov R., eds. *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. 97 of *Proceedings of Machine Learning Research*. PMLR; 2019: 831–840.
54. Liu P, Yuan W, Fu J, Jiang Z, Hayashi H, Neubig G. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *CoRR* 2021; abs/2107.13586.
55. Clark K, Luong M, Le QV, Manning CD. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In: OpenReview.net; 2020.
56. Goodfellow IJ, Pouget-Abadie J, Mirza M, et al. Generative adversarial networks. *Commun. ACM* 2020; 63(11): 139–144. doi: 10.1145/3422622
57. Li J, Monroe W, Jurafsky D. Understanding Neural Networks through Representation Erasure. *CoRR* 2016; abs/1612.08220.
58. Fong RC, Vedaldi A. Interpretable Explanations of Black Boxes by Meaningful Perturbation. In: IEEE Computer Society; 2017: 3449–3457
59. Zhou W, Ge T, Xu K, Wei F, Zhou M. BERT-based Lexical Substitution. In: Korhonen A, Traum DR, Màrquez L., eds. *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*Association for Computational Linguistics; 2019: 3368–3373
60. Bertsimas D, Tsitsiklis J, others . Simulated annealing. *Statistical science* 1993; 8(1): 10–15.
61. Qi G, Shah M. Adversarial Pretraining of Self-Supervised Deep Networks: Past, Present and Future. *CoRR* 2022; abs/2210.13463. doi: 10.48550/arXiv.2210.13463
62. Allamanis M, Sutton C. Mining source code repositories at massive scale using language modeling. In: Zimmermann T, Penta MD, Kim S., eds. *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*IEEE Computer Society; 2013: 207–216
63. Gao T, Yao X, Chen D. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In: Moens M, Huang X, Specia L, Yih SW., eds. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*Association for Computational Linguistics; 2021: 6894–6910
64. Sun Z, Zhang JM, Harman M, Papadakis M, Zhang L. Automatic testing and improvement of machine translation. In: Rothermel G, Bae D., eds. *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*ACM; 2020: 974–985



APPENDIX

A ADVERSARIAL EXAMPLE

We present more OJ cases of adversarial examples generated by MHM and CBA against LSTM in Table A1. These cases are randomly sampled from the 38 examples for human evaluation, without any cherry-picking. MHM in most cases does not consider the textual and contextual information of the original identifier, and randomly substitutes it with a completely unrelated one, e.g., “len” → “month_day_sum”. On the other hand, CBA may be much more natural and consistent, since it mainly performs substitutions such as “m” → “n” and “i” → “index”.

We also list some randomly sampled OJ cases of attacks against CodeBERT-MLM in Table A2, following the same filtering and sampling procedure. Similar findings as above can be found in attacks against CodeBERT-MLM.

We hope these cases may give the readers some intuition about the quality of CBA, compared with MHM.

TABLE A1 Examples of adversarial perturbations against LSTM in OJ.

Original Code	MHM	CBA
<pre> int main() { int n, k, m; scanf("%d %d", &n, &k); if (n == 2 && k == 1){ m = 7; printf("%d\n", m); } else m = (int)(pow(n, n)) - (n - 1) * k; printf("%d\n", m); return 0; } </pre>	n → xueke	n → j
<pre> int main() { char a[20]; int i = 1, n; char c; do { scanf("%s", a); n = strlen(a); if (i == 1) printf("%d", n); else printf(",%d", n); i++; } while ((c = getchar()) != '\n'); } </pre>	a → jinzhi2	n → N i → m
<pre> int main() { char a[7], i, j; scanf("%s", &a); for (i = 1; a[i] != '\0'; i++) j = i; for (i = j; i >= 0; i--) printf("%c", a[i]); return 0; } </pre>	i → wn	i → index

TABLE A1 Examples of adversarial perturbations against LSTM in OJ.

Original Code	MHM	CBA
<pre> int main() { int x, i, result = 1; scanf("%d", &x); if (!x) printf("0"); while (x) { printf ("%d", x%10); x = x / 10; } printf("\n", result); return 0; } </pre>	x → NumberOfJump	x → w
<pre> int main() { int n, a, i; scanf("%d", &n); for (i = 1; i <= 5; i++) { a = n % 10; printf("%d", a); n = (n - a) / 10; if (n == 0) break; } } </pre>	n → left_num	n → m
<pre> int main() { char s[200], w[100]; int i; scanf("%s %s", s, w); for (i = 0; i < 100; i++) if (w + i == strstr(w, s)) { printf ("%d", i); break; } } </pre>	w → index1_tail	w → sw
<pre> int main() { char a [1000]; cin.getline(a, 100); int i; int l = strlen(a); for (i = l - 1; i >= 0; i--) cout << a[i]; cout << endl; } </pre>	i → whatmark	i → l

TABLE A1 Examples of adversarial perturbations against LSTM in OJ.

Original Code	MHM	CBA
<pre> int main () { int len ; char word [30] ; cin >> word ; len = strlen (word) ; cout << len ; while (cin >> word) { len = strlen (word) ; cout << ", " << len ; } cout << endl ; return 0 ; } </pre>	len → month_day_sum	len → count
<pre> int main(void){ char Str[105]; gets(Str); int strl = strlen(Str); for (int i = strl - 1; i >= 0; i--) { if (Str[i] == ' ') { Str[i] = '\0'; printf("%s ", Str + i + 1); } } printf("%s", Str); } </pre>	strl → getnum	i → my
<pre> void p (int n) { int c; cin >> c; if (n != 1) p(n - 1); if (n == 1) cout << c; else cout << ' ' << c; } int main() { int n; cin >> n; p(n); cout << endl; return 0; } </pre>	n → q_head	n → j

TABLE A2 Examples of adversarial perturbations against CodeBERT-MLM in OJ.

Original Code	MHM	CBA
<pre> int main() { int m, i; char a[6]; scanf("%s", a); m = strlen(a); for (i = m; i >= 0; i--) { if (a[i] == '\0') continue; else printf("%c", a[i]); } return 0; } </pre>	m → shuzunan	a → af
<pre> int main() { char c; c = getchar(); while (c != '\n') { if (c == ' ') { cout << " "; while (c == ' ') c = getchar(); } cout << c; c = getchar(); } return 0; } </pre>	c → EVEN	c → j
<pre> int main() { char input[20] = {0}, i = 0; while (scanf("%c", &input[i]) != -1 && input[i] != '\n') i++; for (i--; i >= 0; i--) printf ("%c", input[i]); printf ("\n"); } </pre>	i → means	i → start input → flag

TABLE A2 Examples of adversarial perturbations against CodeBERT-MLM in OJ.

Original Code	MHM	CBA
<pre> int main() { char str[100]; gets(str); int len = strlen(str); for (int i = 0; i < len; i++) { if (str[i] == ' ') { cout << str[i]; while (str[++i] == ' '); } cout << str[i]; } return 0; } </pre>	i → liezuixiao	str → tw
<pre> int main() { char a[7], i, j; scanf("%s", &a); for (i = 1; a[i] != '\0'; i++) j = i; for (i = j; i >= 0; i--) printf ("%c", a[i]); return 0; } </pre>	a → ENDyear	a → ch i → ke j → i
<pre> int main() { char ch; int flag = 0; while ((ch = getchar()) != '\n') { if (ch == ' ' && flag == 0) { putchar(ch); flag = 1; } else if (ch != ' ') { flag = 0; putchar(ch); } } getchar(); getchar(); getchar(); } </pre>	ch → peo2	ch → ah flag → tag