# Code Difference Guided Adversarial Example Generation for Deep Code Models

Zhao Tian
*College of Intelligence and Computing, Tianjin University*
Tianjin, China
tianzhao@tju.edu.cn

Junjie Chen†
*College of Intelligence and Computing, Tianjin University*
Tianjin, China
junjiechen@tju.edu.cn

Zhi Jin
*Key Lab of High Confidence Software Technology, Peking University*
Beijing, China
zhijin@pku.edu.cn

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00149

***Abstract*—Adversarial examples are important to test and enhance the robustness of deep code models. As source code is discrete and has to strictly stick to complex grammar and semantics constraints, the adversarial example generation techniques in other domains are hardly applicable. Moreover, the adversarial example generation techniques specific to deep code models still suffer from unsatisfactory effectiveness due to the enormous ingredient search space. In this work, we propose a novel adversarial example generation technique (i.e., CODA) for testing deep code models. Its key idea is to use code differences between the target input (i.e., a given code snippet as the model input) and reference inputs (i.e., the inputs that have small code differences but different prediction results with the target input) to guide the generation of adversarial examples. It considers both structure differences and identifier differences to preserve the original semantics. Hence, the ingredient search space can be largely reduced as the one constituted by the two kinds of code differences, and thus the testing process can be improved by designing and guiding corresponding equivalent structure transformations and identifier renaming transformations. Our experiments on 15 deep code models demonstrate the effectiveness and efficiency of CODA, the naturalness of its generated examples, and its capability of enhancing model robustness after adversarial fine-tuning. For example, CODA reveals 88.05% and 72.51% more faults in models than the state-of-the-art techniques (i.e., CARROT and ALERT) on average, respectively.**

***Index Terms*—Adversarial Example, Code Model, Guided Testing, Code Transformation**

## I. Introduction

In recent years, deep learning (DL) has been widely used to solve code-based software engineering tasks, such as code clone detection [1] and vulnerability prediction [2] by building DL models based on a large amount of training code snippets (called *deep code models*). Indeed, deep code models have achieved notable performance and largely promoted the process of software development and maintenance [3]–[6]. In particular, some industrial products on deep code models have been released and received extensive attention, such as AlphaCode [7] and Copilot [8].

Like DL models in other areas (e.g., image processing) [9]–[12], the robustness of deep code models is critical [13], [14]. As demonstrated by the existing work [15], [16], adversarial examples are important to test and enhance the model robustness. Specifically, adversarial examples can test a deep code model to reveal faults in it by comparing the prediction results on adversarial examples and that on the original input generating these adversarial examples. Such adversarial examples are called *fault-revealing examples* for ease of presentation, which can be used to augment training data for further enhancing the model robustness. Therefore, improving test effectiveness through generating fault-revealing examples is very important.

However, the existing adversarial example generation techniques in other areas are hardly applicable to deep code models. This is because they tend to perturb an input in continuous space, while the inputs (i.e., source code) for deep code models are discrete. Moreover, source code has to strictly stick to complex grammar and semantics constraints, i.e., the adversarial example generated from an original input should have no grammar errors and preserve the original semantics.

Indeed, some adversarial example generation techniques specific to deep code models have been proposed recently, such as MHM [15], CARROT [13], and ALERT [16]. In general, they share two main steps: (1) designing semantic-preserving code transformation rules, and (2) searching ingredients from the space defined by the rules (i.e., ingredients are the elements required by transformation rules) for transforming an original input (called *target input*) to a semantic-preserving adversarial example. For example, CARROT designs two semantic-preserving code transformation rules (i.e., identifier renaming and dead code insertion), and uses the hill-climbing algorithm to search for the ingredients from the entire space with the guidance of gradients and model prediction changes. ALERT considers the rule of identifier renaming, and uses the naturalness and model prediction changes to guide the ingredient search process.

Although some of them have been demonstrated effective to some degree, they still suffer from major limitations:

- The ingredient space defined by code transformation rules is enormous. For example, all valid identifier names could be the ingredients for identifier renaming transformation. Hence, searching for the ingredients that facilitate generating fault-revealing examples is challenging. The existing techniques tend to utilize model prediction changes after performing transformations on the target

†Junjie Chen is the corresponding author.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00149

input greedily to guide the search process, which is likely to fall into local optimum and thus limits test effectiveness [16].
- Frequently invoking the target model could affect test efficiency via adversarial example generation [13], [16], as model invocation is the most costly part during testing. However, the existing techniques often involve frequent model invocations due to calculating gradients or guiding the search direction via model prediction.
- Developers care about natural semantics of code since it can assist human comprehension on detected faults in models [17]. Hence, ensuring the naturalness of generated adversarial examples is important. However, all the existing techniques (except ALERT [16]) do not consider this factor. For example, CARROT designs the rule of dead code insertion, but it may largely damage the naturalness of generated examples (especially when a large amount of dead code is inserted).

Overall, a more effective adversarial example generation technique for testing deep code models should improve the ingredient search process, and guarantee the times of model invocations as few as possible and the naturalness of generated adversarial examples as much as possible. Our work proposes such a technique, called **CODA** (**CO**de **D**ifference guided **A**dversarial example generation).

To improve test effectiveness, the key idea of CODA is to take *reference inputs* (that have small code differences with the target input but have different prediction results) as the guidance for generating adversarial examples. This is an innovative perspective and closely utilizes the unique characteristics of deep code models. With reference inputs, the ingredient space can be largely reduced. Specifically, reference inputs can be regarded as *invalid* fault-revealing adversarial examples generated from the target input, where "invalid" refers to altering the original semantics and "fault-revealing" refers to producing different prediction results. That is, the code differences brought by reference inputs over the target input contribute to the generation of invalid but *fault-revealing* example to a large extent. Hence, if we extract the ingredients from the code differences to support semantic-preserving transformations on the target input, their code differences can be gradually reduced without altering the original semantics, and thus a *valid* fault-revealing example is likely to be generated. In this way, the ingredient space is reduced as the one constituted by only code differences between reference inputs and the target input, and thus the search process can be improved.

Based on the key idea, CODA considers code structure differences and identifier differences to support ingredient extraction for equivalent structure transformations and identifier renaming transformations. It can preserve the original semantics during the generation process. Equivalent structure transformations (e.g., transforming a `for` loop to an equivalent `while` loop) do not affect the naturalness of generated examples, and thus CODA first applies this kind of transformations to reduce code differences for generating adversarial examples. Then, identifier renaming transformations are applied to further reduce code differences to improve test effectiveness. To ensure the naturalness of generated examples by this kind of transformations, CODA measures semantic similarity between identifiers for guiding iterative transformations. Note that we do not emphasize the novelty in these transformation rules since some of them have been proposed before, and the main novelty lies in the code-difference-guided transformation process in CODA, which is the key to improve the test effectiveness with these transformations. In particular, CODA just involves necessary model invocations to check whether the generated example reveals a fault, without extra gradient calculation and a large amount of model prediction for guiding the search process.

We conducted an extensive study to evaluate CODA based on three popular pre-trained models (i.e., CodeBERT [5], GraphCodeBERT [6], and CodeT5 [18]) and five code-based tasks. In total, we used 15 subjects. Our results demonstrate the effectiveness and efficiency of CODA. For example, on average across all the subjects, CODA revealed 88.05% and 72.51% more faults in models than the two state-of-the-art adversarial example generation techniques (i.e., CARROT [13] and ALERT [16]), respectively. The time spent by CODA on completing the testing process for the 15 subjects is 196.96 hours, while those by CARROT and ALERT are 290.87 hours and 374.51 hours, respectively. Furthermore, we investigated the value of the generated adversarial examples by using them to enhance the robustness of the target model via an adversarial fine-tuning strategy. The results show that the models after fine-tuning with the examples generated by CODA can reduce 62.19%, 65.67%, and 73.95% of faults revealed by CARROT, ALERT, and CODA on average, respectively.

To sum up, our work makes four major contributions:
- **Novel Perspective**. We propose a novel perspective of utilizing code differences between reference inputs and the target input to guide the fault-revealing example generation process for testing deep code models.
- **Tool Implementation**. We implement CODA following the novel perspective by (1) measuring code structure and identifier differences and (2) designing and guiding corresponding semantic-preserving code transformations.
- **Performance Evaluation**. We conducted an extensive study on three popular pre-trained models and five code-based tasks, demonstrating the effectiveness and efficiency of CODA over two state-of-the-art techniques.
- **Public Artifact**. We released all the experimental data and our source code at the project homepage [19] for experiment replication, future research, and practical use.

## II. Background and Motivation

### A. Deep Code Models

DL has been widely used to process source code [2], [20]–[25]. Some pre-trained DL models have been built based on a large number of code snippets, among which CodeBERT [5], GraphCodeBERT [6], and CodeT5 [18] are three state-of-the-art pre-trained models. The pre-trained models have brought

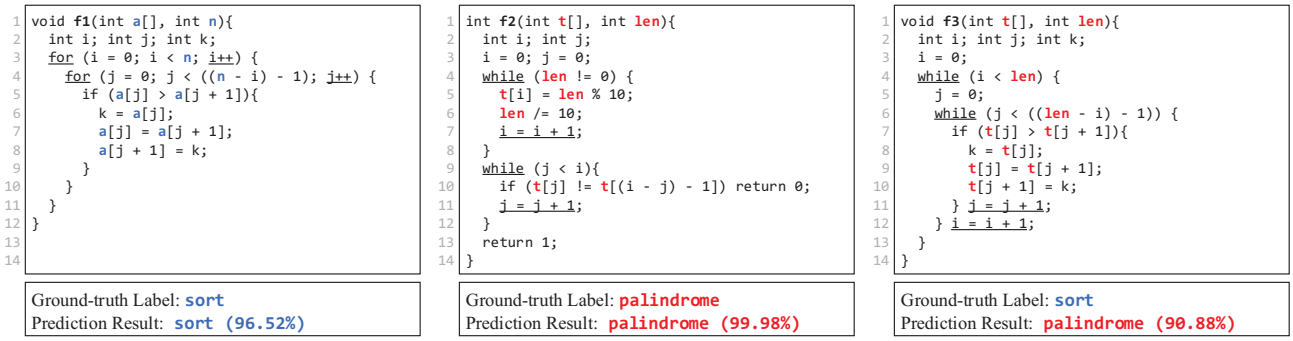



Fig. 1. An illustrating example (the target input *f1*, a reference input *f2*, and a fault-revealing adversarial example *f3* generated from *f1*)

breakthrough changes to many code-based tasks [26], including both classification and generation tasks, by fine-tuning them on the datasets of the corresponding tasks. The former makes classification based on the given code snippets (e.g., vulnerability prediction [2]), while the latter produces a sequence of information based on code snippets or natural language descriptions (e.g., code completion [21]). Following most of the existing work on generating adversarial examples for deep code models [13], [15], [16], our work also focuses on the classification tasks and takes the generation tasks (targeted by ACCENT [27], CCTest [14], etc.) as our future work. In our study, we adopted all the five classification tasks used in the studies of evaluating the state-of-the-art adversarial example generation techniques (i.e., CARROT [13] and ALERT [16]).

### B. Problem Definition

Given a code snippet $x$ that is processed as the required format by the target deep code model $\mathcal{M}$ (e.g., abstract syntax trees required by code2seq [3], control-flow graphs required by DGCNN [28], or data-flow graphs required by GraphCodeBERT [6]), $\mathcal{M}$ can predict a probability vector for $x$, each element in which represents the probability classifying $x$ to the corresponding class. The class with the largest probability is the final prediction result of $\mathcal{M}$ for $x$. If the prediction result is different from the ground-truth label (denoted as $y$) of $x$, it means that $\mathcal{M}$ makes a wrong prediction on $x$; otherwise, $\mathcal{M}$ makes a correct prediction.

Same as the existing work [13], [16], our goal is to improve test effectiveness through more effectively generating fault-revealing examples, which subsequently can be used to enhance model robustness. The existing techniques specific to deep code models always generate adversarial examples from a target input by performing a series of semantic-preserving code transformations [13], [15], [16], which is also followed by our work. For ease of understanding, we formally present our problem as finding $x'$ ($x' \in \epsilon \wedge y = \mathcal{M}(x) \neq \mathcal{M}(x')$) from a target input $x$ for the target model $\mathcal{M}$. Here, $\epsilon$ is the universal set of code snippets that satisfy the grammar constraints and preserve the semantics of $x$. $y = \mathcal{M}(x)$ means that we just regard the test inputs on which $\mathcal{M}$ makes correct predictions as target inputs since analyzing robustness upon such inputs is more meaningful following the existing work [13], [16], where $\mathcal{M}(x)$ refers to the prediction result of $\mathcal{M}$ on $x$. $\mathcal{M}(x) \neq \mathcal{M}(x')$ means that $x'$ reveals a fault in $\mathcal{M}$, that is, it is a fault-revealing example generated from $x$. Besides, an effective adversarial example generation technique should be *efficient* to find $x'$ and ensure the *naturalness* of $x'$ (i.e., natural to human comprehension [16]), which are indeed carefully considered by our proposed technique.

### C. Motivating Example

We use a real-world example (simplified for ease of illustration) to motivate our key idea: utilizing code differences between reference inputs and the target input to guide the generation of adversarial examples. In Figure 1, the first code snippet *f1* is the target input from the test set of the functionality classification task [29], and the two state-of-the-art techniques (i.e., CARROT [13] and ALERT [16]) do not generate fault-revealing examples from it for the deep code model CodeBERT since they can fall into local optimum in the enormous ingredient space. The second code snippet *f2* is a reference input from the training set of this task, which has the different label with *f1*.

As presented before, *f2* can be regarded as an *invalid* fault-revealing example from *f1*, as they are semantically inconsistent but have different prediction results. The code differences between *f1* and *f2* mainly contribute to this phenomenon. From this perspective, to generate a *valid* fault-revealing example (denoted as *f3*) from *f1*, we should perform semantic-preserving code transformations on *f1*, and the transformations should reduce the code differences between *f1* and *f2* to alter the prediction result of the model from *f1*. That is, the ingredients supporting these transformations should be extracted from the code differences brought by *f2*. With the guidance of code differences, by performing equivalent structure transformations on *f1* (i.e., transforming `for` loops to `while` loops, where `while` loops are the used loop structure in *f2*) and identifier remaining transformations (i.e., renaming `a` and `n` to `t` and `len` respectively, where `t` and `len` are the used identifier names in *f2*), *f3* is generated as shown in the third code snippet in Figure 1 and indeed reveals a fault in the model, i.e., making a wrong prediction (`palindrome`) with a high confidence (90.88%).

Based on the code differences between the target input and the reference input, the ingredient space is largely reduced. For

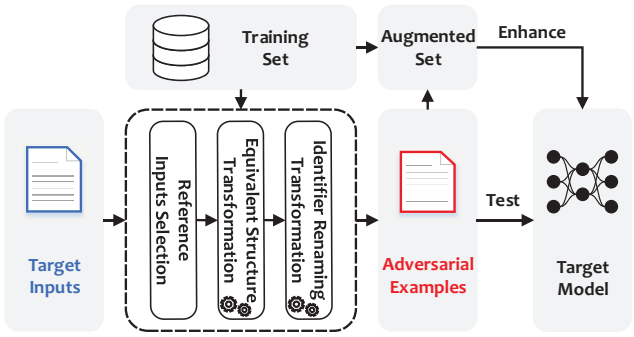



Fig. 2. Overview of CODA.

example, the ingredient space defined by identifier renaming transformations is reduced from all valid identifier names (i.e., almost infinite) to the identifier names occurring in the reference input but not in the target input (i.e., only two identifiers in this example). Hence, it helps improve the ingredient search process and thus improves test effectiveness. On the other hand, too small ingredient space may lose too many ingredients useful to generate fault-revealing examples, and thus we will select a set of reference inputs (rather than only one reference input) for guiding the generation process so as to balance the ingredient-space size and the number of useful ingredients.

## III. Approach

### A. Overview

We propose a novel perspective to generate adversarial examples for better testing deep code models, which utilizes code differences between reference inputs and the target input to guide the generation process. From this perspective, we design an effective adversarial example generation technique, called CODA. Specifically, the code differences brought by reference inputs provide effective ingredients for altering the prediction result of the target input by transforming it with these ingredients, which can contribute to the generation of fault-revealing examples in CODA. However, as the semantics of reference inputs and the target input are different, the ingredients from some kinds of code differences can alter the original semantics, which is not allowed by the adversarial examples for deep code models. Hence, in CODA, we consider structure differences and identifier differences for measuring code differences between them, which can preserve the original semantics in generation. In this way, the ingredient space can be reduced as the one constituted by the two kinds of code differences, and thus the ingredient search process (for generating adversarial examples) can be largely improved.

In fact, not all inputs that have different prediction results with the target one, can be regarded as effective reference inputs for improving test effectiveness. In other words, different inputs could have different degrees of capabilities for reducing the ingredient search space and providing effective ingredients for altering the prediction result of the target input. Hence, the first step in CODA is to select effective reference

TABLE I
Descriptions of Equivalent Structure Transformations

| Transformation | Description |
|---|---|
| $R_1$-*loop* | equivalent transformation among `for` structure and `while` structure |
| $R_2$-*branch* | equivalent transformation between `if-else(-if)` structure and `if-if` structure |
| $R_3$-*calculation* | equivalent numerical calculation transformation, e.g., `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `\|=` , `^=` |
| $R_4$-*constant* | equivalent transformation between a constant and a variable assigned by the same constant |

inputs for the target input in order to improve test effectiveness (Section III-B). Based on the selected reference inputs, CODA then measures structure differences and identifier differences over the target input, which support extracting the ingredients for two corresponding kinds of semantic-preserving code transformations (i.e., equivalent structure transformations and identifier renaming transformations). With the guidance of reducing their code differences, the target input could be effectively transformed to a fault-revealing example based on the two kinds of transformations. As equivalent structure transformations do not affect the naturalness of generated examples, CODA first applies this kind of transformations to reduce code differences for improving the generation of fault-revealing examples (Section III-C). Then, we apply identifier renaming transformations to further reduce the code differences for improving the test effectiveness (Section III-D).

Figure 2 shows the overview of CODA. Due to the smaller ingredient search space (but including effective ingredients) and clearer generation direction (towards the direction of reducing code differences without altering the original semantics), the test effectiveness could be improved by CODA.

### B. Reference Inputs Selection

The goal of reference inputs is to reduce the ingredient space. The reduced space should include the ingredients that are effective to transform the target input to a fault-revealing example. In this way, the adversarial example generation process can be largely improved by searching for effective ingredients more efficiently. Although all the inputs that have different prediction results with the target one can provide ingredients for altering the prediction result of the target one after transformations, their capabilities for adversarial example generation can be different. To transform the target input to a fault-revealing example with fewer perturbations, CODA should select the reference inputs, which provide the ingredients that are more likely to generate a fault-revealing example on the target input. Similar to the existing work [30]–[33], we assume the prediction result of the target input is more likely to be changed from its original class denoted as $c_i$ (with the largest probability predicted by the target model) to the class with the second largest probability (denoted as $c_j$). Hence, the ingredients in the inputs belonging to $c_j$ are more likely to generate a fault-revealing example on the target input, and thus CODA selects the inputs belonging to $c_j$ as the

initial set of reference inputs. Note that all the reference inputs are selected from the training set to (1) avoid introducing the contents beyond the cognitive scope of the model and (2) ensure the sufficiency of the inputs belonging to $c_j$. Moreover, we only consider the training inputs whose prediction results are consistent with their ground-truth labels so as to avoid introducing noise.

However, the number of inputs belonging to the same class (i.e., $c_j$ as above) could be large, and thus the ingredient space constituted by code differences between them and the target input could be also large. Hence, to further reduce the ingredient space for more effective adversarial example generation, CODA selects a subset of inputs with high similarity to the target input from the initial set of reference inputs, as the final set of reference inputs used by CODA. This is because smaller code differences can effectively limit the number of ingredients, leading to smaller ingredient space. CODA does not select only one reference input, as too small ingredient space could incur a high risk of missing too many ingredients contributing to fault-revealing example generation.

We further introduce how to measure the similarity between the target input (denoted as $t$) and a reference input (denoted as $r$) for the above selection. In general, we can adopt some pre-trained models to represent the code as a vector and then measure code similarity by calculating the vector distance, like many existing studies [4], [5], [34]. However, as presented in Section III-A, CODA first applies equivalent structure transformations (rather than identifier renaming transformations) to reduce code differences for adversarial example generation. Moreover, the identifiers used in different code snippets are usually different due to the enormous identifier space, which may lead to the low similarity between various code snippets. Hence, when measuring code similarity, CODA eliminates the influence of identifiers by replacing them with the placeholder <unk>. Specifically, CODA first represents $t$ and $r$ after placeholder replacement as vectors respectively based on CodeBERT [5] (one of the most widely-used pre-trained models [35]–[37]), and then calculates the cosine similarity between vectors. As the descending order of the calculated similarity, CODA selects Top-$N$ reference inputs for the follow-up generation process. Note that to make the selection process efficient, we randomly sampled $U$ inputs from the initial set for this step of selection. We will study the influence of both $U$ and $N$ on CODA in Section VI.

### C. Equivalent Structure Transformation

Based on the small set of selected reference inputs and the guidance of code differences, CODA first reduces structure differences by applying equivalent structure transformations to the target input.

To preserve the semantics of the target input, we design four categories of equivalent structure transformations (without affecting code naturalness) in CODA following the existing work in metamorphic testing and code refactoring [38]–[40]. In particular, we systematically consider all common kinds of code structures, i.e., *loop* structures, *branch* structures, and *sequential* structures (including numerical calculation and constant usage). We explain the four categories in detail in Table I. For each category of transformations, it may include several specific rules. For example, the rules of transformation on `+=` and transformation on `-=` belong to the category of $R_3$-*calculation*, and the rules of transforming `for` loop to `while` loop and transforming `while` loop to `for` loop belong to the category of $R_1$-*loop*. In total, CODA has 20 specific rules for the four categories of transformations. Due to the space limit, we list all these specific rules at our project homepage [19]. These rules are general to the vast majority of popular programming languages (e.g., C, C++, and Java), which ensures the generality of CODA to a large extent. Note that in $R_4$-*constant*, the newly-defined variable cannot be the same as the existing variables in the code; otherwise, it may incur grammar errors and alter the original semantics.

Then, we illustrate how to apply these rules for reducing code differences. Each rule involves two structures, i.e., the one before transformation ($s_b$) and the one after transformation ($s_a$). CODA first counts the occurring times of $s_b$ and $s_a$ in the set of selected reference inputs (denoted as $n_b$ and $n_a$), and then calculates their occurring distribution, i.e., $\frac{n_b}{n_b+n_a}$ and $\frac{n_a}{n_b+n_a}$. Further, CODA applies each rule in a probabilistic way to reduce the occurring distribution differences in terms of $s_b$ and $s_a$ between reference inputs and the target input since probabilistic methods tend to improve effectiveness compared to deterministic methods in general [41]. In this way, the structure differences in terms of $s_b$ and $s_a$ can be reduced. Specifically, for each occurrence of $s_b$ in the target input, CODA applies this rule with the probability of $\frac{n_a}{n_b+n_a}$, also indicating it can be retained with the probability of $\frac{n_b}{n_b+n_a}$.

In this step, CODA obtains $M$ inputs from the target input, each of which is generated by applying all the applicable rules together on the target input in the above probabilistic way, and then selects the input with the highest average similarity (also measured by the method described in Section III-B) to the selected reference inputs as the one for the follow-up generation process.

### D. Identifier Renaming Transformation

To facilitate the generation of fault-revealing examples, CODA then applies identifier renaming transformations to further reduce code differences. Inspired by the existing work [13], [15], [16], identifier renaming transformation in CODA refers to replacing the name of an identifier in the target input with the name of an identifier in the selected reference inputs. For ease of presentation, we denote the set of identifiers in the target input as $V_t$ and the set of identifiers in the selected reference inputs as $V_r$. To preserve the semantics of the target input and ensure the grammatical correctness of the generated example, CODA ensures that the identifiers used for replacement do not exist in the target input.

Then, we illustrate how to apply this kind of transformations to the input obtained from the last step. As demonstrated by the existing work [13], [15], [16], renaming identifiers is effective to generate fault-revealing examples, but can negatively affect

naturalness of generated examples. To ensure the naturalness of generated examples, CODA considers the semantic similarity between identifiers and designs an iterative transformation process like ALERT [16]. Specifically, CODA measures the semantic similarity between each identifier in $V_t$ and each identifier in $V_r$ by representing each identifier as a vector via word embedding. Here, CODA builds the pre-trained language model with FastText [42] and calculates the cosine similarity between vectors to measure their semantic similarity. Then, CODA prioritizes each pair of identifiers in the descending order of their semantic similarity, and iteratively applies this transformation based on each pair of identifiers in the ranking list, which ensures more natural transformations can be first performed. CODA ensures the pair of identifiers will not introduce repetitive identifiers in the generated example in each iteration; otherwise, this pair will be discarded. After each iteration, CODA invokes the target model to check whether a fault is detected by the generated example.

Following the existing work [13], [16], the iterative generation process terminates until a fault-revealing example from the target input is generated or all the pairs are used by this transformation. This is because more fault-revealing examples generated from the same target input tend to detect duplicate faults in the model. The setting can be adjusted by users according to the demand in practice. In this work, we directly adopt the setting from the existing work [13], [16] in CODA.

Overall, CODA only invokes the target model when checking if a fault is detected by the generated example, which is necessary for testing the model. Hence, CODA can largely reduce the number of model invocations compared with the existing techniques, confirmed by our study (Section V-A).

## IV. Evaluation Design

In the study, we address four research questions (RQs):

- **RQ1:** How does CODA perform in terms of effectiveness and efficiency compared with state-of-the-art techniques?
- **RQ2:** Are the adversarial examples generated by CODA useful to enhance the robustness of deep code models?
- **RQ3:** Does each main component contribute to the overall effectiveness of CODA?
- **RQ4:** Are the adversarial examples generated by CODA natural for humans?

### A. Subjects

*1) Datasets and Tasks:* To sufficiently evaluate CODA, we consider all the five code-based tasks in the studies of evaluating state-of-the-art techniques (i.e., CARROT [13] and ALERT [16]). The statistics of datasets are shown at the first four columns in Table II, each of which represents the task, the number of inputs in the training/validation/test set, the number of classes for the classification task, and the programming language for the inputs.

The task of *vulnerability prediction* aims to predict whether a given code snippet has vulnerabilities. Its used dataset is extracted from two C projects [2]. The task of *clone detection* aims to detect whether two given code snippets are equivalent in semantics. Its used dataset is from BigCloneBench [43], the most widely-used dataset for clone detection. The task of *authorship attribution* aims to identify the author of a given code snippet. Its used dataset is the Google Code Jam (GCJ) dataset [44], which contains 660 Python code files and 66 author information after removing 40 code files in other programming languages [16]. The task of *functionality classification* aims to classify the functionality of a given code snippet. Its used dataset is the Open Judge (OJ) benchmark [45]. The task of *defect prediction* aims to predict whether a given code snippet is defective and its defect type. Its used dataset is the CodeChef dataset [46].

TABLE II
Statistics of our used subjects

| Task | Train/Val/Test | Class | Language | Model | Acc. |
|---|---|---|---|---|---|
| Vulnerability Prediction | 21,854/2,732/2,732 | 2 | C | CodeBERT | 63.76% |
| | | | | GCBERT | 63.65% |
| | | | | CodeT5 | 63.83% |
| Clone Detection | 90,102/4,000/4,000 | 2 | Java | CodeBERT | 96.97% |
| | | | | GCBERT | 97.36% |
| | | | | CodeT5 | 98.08% |
| Authorship Attribution | 528/–/132 | 66 | Python | CodeBERT | 90.35% |
| | | | | GCBERT | 89.48% |
| | | | | CodeT5 | 92.30% |
| Functionality Classification | 41,581/–/10,395 | 104 | C | CodeBERT | 98.18% |
| | | | | GCBERT | 98.66% |
| | | | | CodeT5 | 98.79% |
| Defect Prediction | 27,058/–/6,764 | 4 | C/C++ | CodeBERT | 84.37% |
| | | | | GCBERT | 83.98% |
| | | | | CodeT5 | 81.54% |

* GCBERT is short for GraphCodeBERT.

*2) Models:* Same as the existing work [16], we used the state-of-the-art pre-trained models, i.e., CodeBERT [5] and GraphCodeBERT [6], in our study, and also used the more recent CodeT5 [18]. Several alternative pre-trained models have been studied in the experiments of CARROT (such as LSTM and TBCNN). However, these models did not demonstrate superior performance compared to CodeBERT, GraphCodeBERT, and CodeT5. As a result, our investigation in this study focused on the latter three models.

We fine-tuned them on the five tasks based on the corresponding datasets, respectively. When fine-tuning CodeBERT and GraphCodeBERT on these tasks (except GraphCodeBERT on functionality classification and defect prediction), we used the same hyper-parameter settings provided by the existing work [13], [16]. As there is no instruction on the hyper-parameter settings for fine-tuning GraphCodeBERT on functionality classification and defect prediction, we used the same settings as the one used by authorship attribution (they are all multi-class classification tasks). Indeed, the achieved model performance outperforms that achieved by the models (e.g., TBCNN [45] and CodeBERT [5]) used in the existing work [13] on the same datasets [13], indicating that the transferred hyper-parameter settings are reasonable. Similarly, we used the same settings for CodeT5 as CodeBERT and the achieved performance of CodeT5 is better than (or close to) those achieved by CodeBERT and GraphCodeBERT on all the tasks. The detailed settings can be found at our project

homepage [19]. In total, we obtained 15 deep code models as the subjects. The last two columns in Table II show the used pre-trained model and the accuracy of the deep code model after fine-tuning on the corresponding task, respectively.

Overall, the subjects used in our study are diverse, involving different tasks, different pre-trained models, different numbers of classes, different programming languages, etc. It is very helpful to sufficiently evaluate the performance of CODA.

### B. Compared Techniques

In the study, we compared CODA with two state-of-the-art adversarial example generation techniques for deep code models, i.e., **CARROT** [13] and **ALERT** [16], which have been introduced in Section I (the third paragraph). We adopted their implementations and the recommended parameter settings provided by the corresponding papers [13], [16]. As the original version of CARROT can only support C/C++ code, we extended it to Python and Java code for sufficient comparison.

### C. Implementations

We implemented CODA in Python and adopted tree-sitter [47] to extract identifiers from code following the existing work [16]. We set the parameters in CODA by conducting a preliminary experiment, i.e., $U = 256$ (the number of sampled inputs from the initial set for similarity calculation), $N = 64$ (the number of reference inputs selected after similarity calculation), and $M = 64$ (the number of examples generated via structure transformations). We will discuss the influence of the settings of main parameters in Section VI. All the experiments were conducted on a server with an Ubuntu 20.04 system with Intel(R) Xeon(R) Silver 4214 @ 2.20GHz CPU, and NVIDIA GeForce RTX 2080 Ti GPU.

## V. Results and Analysis

### A. RQ1: Effectiveness and Efficiency

*1) Setup:* For each deep code model, we applied CODA, CARROT, and ALERT to generate adversarial examples from each target input in the test set to test it, respectively. We measured their effectiveness and efficiency based on the following metrics. To reduce the influence of randomness, we repeated all the experiments (including those for other RQs) 10 times, and reported the average results.

We first measured the number of revealed faults by each technique. As presented in Section III-D, the faults revealed by several adversarial examples from the same target input tend to be duplicate as claimed in the existing work [48], [49]. Hence, CODA produces *at most one* fault-revealing example for each target input same as the generation process of ALERT and CARROT. That is, when a fault-revealing example is generated for a given target input, it will move to the next target input. Therefore, the number of revealed faults is equal to the number of targets inputs from which a fault-revealing example is generated here. Since the sizes of different test sets are different, we reported **the rate of revealed faults (RFR)**, instead of the number of revealed faults, for better presentation, same as the existing work [13], [48]. The rate of revealed faults for each subject refers to the ratio of the number of revealed faults to the total number of target inputs (that are correctly predicted as mentioned in Section II-B) in the test set of the subject. Larger RFR values mean better test effectiveness.

Also, it is important to measure whether the prediction confidence (i.e., the probability of being the ground-truth class of the target input) is decreased by the generated examples (although there is no fault-revealing example generated from a target input). Reducing prediction confidence indicates that the generated examples make the model less robust. Hence, we calculated **prediction confidence decrement (PCD)** to measure the effectiveness of each technique. PCD is calculated by the prediction confidence of the target input minus the minimum prediction confidence of the set of generated examples from the target input. If the former is smaller than the latter, we regard PCD to be 0, indicating that the generated examples cannot decrease the prediction confidence of the target input. Larger PCD values mean better test effectiveness.

Following the existing work [13], [16], we used the **time spent on the overall testing process** (i.e., completing the testing process for all the subjects) and **the average number of model invocations for generating examples from a target input**, to measure the efficiency of each technique. Less time and fewer model invocations mean higher test efficiency.

*2) Results:* Table III shows the comparison results among CARROT, ALERT, and CODA in terms of RFR. From this table, CODA always outperforms CARROT and ALERT on all the subjects, demonstrating the stable effectiveness of CODA. On average, CODA improves 70.11% and 89.83% higher RFR than CARROT and ALERT across all the five tasks on CodeBERT, 89.34% and 57.67% higher RFR on GraphCodeBERT, and 109.26% and 73.12% higher RFR on CodeT5, respectively.

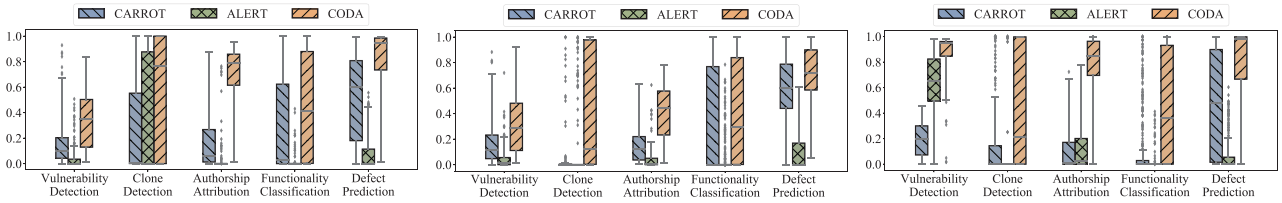
We then investigated the *unique value* of each technique by analyzing their overlap on target inputs where fault-revealing examples are generated. On average across all the subjects, there are 30.68% target inputs where only CODA generates fault-revealing examples among the three techniques, while there are just 7.48% and 2.88% target inputs where only CARROT and ALERT generate fault-revealing examples, respectively. The results demonstrate that CODA has the largest unique value in revealing faults in deep code models among the three techniques.

We analyzed the effectiveness of CODA on *different lengths of code snippets* (ranging from 3 to 8,148 across the five datasets). We measured the Spearman correlation [50] between code-snippet length and RFR of CODA, and the coefficient is 0.17 (p-value $< 0.001$). That is, there is a weak positive correlation between them. That indicates the test effectiveness of CODA is not significantly affected by code-snippet length, even slightly better on larger code snippets in statistics.

Figures 3(a), 3(b), and 3(c) show the comparison results among the three techniques in terms of PCD on CodeBERT, GraphCodeBERT, and CodeT5, respectively. From

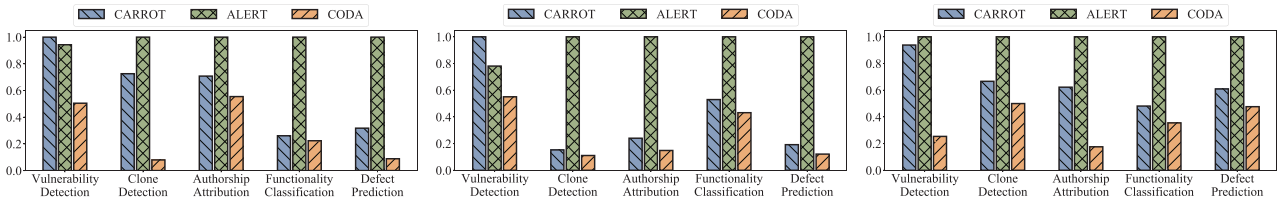
TABLE III
EFFECTIVENESS COMPARISON IN TERMS OF RFR

| Task | CodeBERT | | | GraphCodeBERT | | | CodeT5 | | |
|---|---|---|---|---|---|---|---|---|---|
| | CARROT | ALERT | CODA | CARROT | ALERT | CODA | CARROT | ALERT | CODA |
| Vulnerability Prediction | 33.72% | 53.62% | **89.58%** | 37.40% | 76.95% | **94.72%** | 84.32% | 82.69% | **98.87%** |
| Clone Detection | 20.78% | 27.79% | **44.65%** | 3.50% | 7.96% | **27.37%** | 12.89% | 14.29% | **42.07%** |
| Authorship Attribution | 44.44% | 35.78% | **79.05%** | 31.68% | 61.47% | **92.00%** | 20.56% | 66.41% | **97.17%** |
| Functionality Classification | 44.15% | 10.04% | **56.74%** | 42.76% | 11.22% | **57.44%** | 38.26% | 35.37% | **78.07%** |
| Defect Prediction | 71.59% | 65.15% | **95.18%** | 79.08% | 75.87% | **96.58%** | 38.26% | 35.37% | **78.07%** |
| Average | 42.94% | 38.48% | **73.04%** | 38.88% | 46.69% | **73.62%** | 33.91% | 40.99% | **70.96%** |




(a) PCD on testing CodeBERT (b) PCD on testing GraphCodeBERT (c) PCD on testing CodeT5

Fig. 3. Comparison in terms of prediction confidence decrement




(a) Model invocations on testing CodeBERT (b) Model invocations on testing GraphCodeBERT (c) Model invocations on testing CodeT5

Fig. 4. Comparison in terms of model invocations (y-axis shows the normalized values following the existing work [16])

these figures, the upper quartile, median, and lower quartile of CODA are always larger than (or equal to) those of both CARROT and ALERT regardless of the tasks and the pre-trained models, demonstrating that the adversarial examples generated by CODA can decrease model prediction confidence more significantly. For example, on CodeBERT, the average improvements of CODA over CARROT and ALERT are 101.88% and 520.65% across all the tasks in terms of average PCD, respectively. Similarly, CODA improves 76.35% and 560.15% higher PCD than CARROT and ALERT on GraphCodeBERT, and 389.10% and 397.08% higher PCD on CodeT5, respectively.

Besides, CARROT, ALERT, and CODA take 290.87 hours, 374.51 hours, and 196.96 hours to complete the entire testing process on all subjects, respectively. Further, we measured the number of model invocations for each target input during the testing process, whose results are shown in Figures 4(a), 4(b), and 4(c). From these figures, CODA performs fewer model invocations than both CARROT and ALERT regardless of the tasks and pre-trained models. On average, CODA performs 65.73% and 78.58% fewer model invocations than CARROT and ALERT across all the tasks on CodeBERT, 34.07% and 75.31% fewer model invocations on GraphCodeBERT, and 52.97% and 70.09% fewer model invocations on CodeT5, respectively. The results demonstrate that CODA has the significantly highest efficiency among the three techniques.

Overall, the guidance of code differences in CODA largely improve test effectiveness and efficiency, which is also the reason why ALERT and CARROT underperform CODA. Besides, the reason why ALERT usually outperforms CARROT may lie in the former searches more sufficiently, which is confirmed by more model invocations made by ALERT as above.

> **Answer to RQ1:** CODA takes less time with fewer model invocations on completing the entire testing process, but generates more fault-revealing examples with more significant prediction confidence decrement on all the subjects, than the state-of-the-art baselines.

### B. RQ2: Model Robustness Enhancement

*1) Setup:* We studied the value of generated adversarial examples by using them to enhance the robustness of the target model via an adversarial fine-tuning strategy. For each subject, we divided the test set into two equal parts ($S_1$ and $S_2$), to avoid data leakage between the augmented training set and the evaluation set constructed by the same technique. Specifically, we applied each technique to generate examples from $S_1$, and obtained an fault-revealing example or an example that produces the largest decrement on prediction confidence (if

TABLE IV
ROBUSTNESS ENHANCEMENT OF THE TARGET MODELS AFTER ADVERSARIAL FINE-TUNING

| Task | Model | Ori | | | CARROT | | | ALERT | | | CODA | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | CARROT | ALERT | CODA | CARROT | ALERT | CODA | CARROT | ALERT | CODA | CARROT | ALERT | CODA |
| Vulnerability Prediction | CodeBERT | 62.96% | 62.77% | **63.03%** | 29.14% | 21.11% | **29.69%** | 23.43% | 26.27% | **34.44%** | 32.16% | 31.73% | **38.82%** |
| | GraphCodeBERT | **62.99%** | 62.88% | 62.92% | 12.37% | 19.59% | **21.65%** | 16.33% | 17.35% | **23.71%** | 25.77% | 24.74% | **34.02%** |
| | CodeT5 | 63.69% | 63.81% | **63.92%** | 52.03% | 39.76% | **82.03%** | 42.26% | **49.11%** | 44.26% | 41.43% | 45.52% | **52.54%** |
| Clone Detection | CodeBERT | 97.39% | 96.45% | **97.45%** | 83.15% | 42.31% | **94.44%** | 52.65% | 72.46% | **75.32%** | 38.51% | 71.45% | **89.78%** |
| | GraphCodeBERT | 97.01% | 97.22% | **97.43%** | 75.00% | 66.67% | **77.50%** | 79.17% | 84.29% | **92.31%** | 35.71% | 57.69% | **92.97%** |
| | CodeT5 | 97.73% | 97.14% | **98.10%** | 67.77% | 57.63% | **75.85%** | 69.94% | 64.36% | **81.63%** | 42.15% | 51.74% | **79.88%** |
| Authorship Attribution | CodeBERT | 90.55% | 89.39% | **90.91%** | **45.06%** | 40.67% | 41.03% | 51.25% | 56.25% | **58.82%** | 45.67% | 43.33% | **76.47%** |
| | GraphCodeBERT | 89.39% | 88.72% | **90.35%** | **81.75%** | 67.08% | 72.40% | 79.41% | 78.67% | **100.00%** | 45.59% | 80.39% | **84.75%** |
| | CodeT5 | 92.43% | 92.68% | **93.03%** | 70.95% | 65.91% | **73.48%** | 55.73% | 71.88% | **76.44%** | 44.31% | 52.56% | **72.37%** |
| Functionality Classification | CodeBERT | 98.11% | 98.52% | **98.56%** | **83.46%** | 72.80% | 81.51% | 70.83% | 71.75% | **79.41%** | 78.92% | 71.18% | **95.43%** |
| | GraphCodeBERT | 98.48% | 98.55% | **98.72%** | 67.53% | 75.19% | **77.27%** | 32.04% | 52.62% | **62.98%** | 91.22% | 90.81% | **93.08%** |
| | CodeT5 | 97.92% | 98.46% | **98.63%** | 25.31% | 21.33% | **27.36%** | 41.07% | 57.14% | **57.42%** | 24.87% | 59.58% | **63.76%** |
| Defect Prediction | CodeBERT | 83.50% | 84.16% | **84.44%** | 52.73% | 25.81% | **66.03%** | 74.88% | 75.87% | **83.12%** | 76.86% | 68.66% | **85.36%** |
| | GraphCodeBERT | 83.34% | 84.00% | **84.53%** | 68.20% | 48.54% | **74.88%** | 52.73% | **63.91%** | 59.45% | 67.08% | 68.66% | **76.14%** |
| | CodeT5 | 80.92% | 81.32% | **81.57%** | 31.48% | 34.08% | **37.73%** | 31.75% | 42.22% | **55.77%** | 54.45% | 54.18% | **73.83%** |
| Average | | 86.43% | 86.40% | **86.91%** | 56.40% | 46.57% | **62.19%** | 51.56% | 58.94% | **65.67%** | 49.65% | 58.15% | **73.95%** |

no fault-revealing example is generated) for each target input. These examples were integrated with the training set to form the augmented training set, which is used for fine-tuning the model. Hence, for a given subject, the size of the augmented training set constructed by each technique is the same.

After obtaining a fine-tuned model for each subject with each technique, we evaluated it on the evaluation set of the fault-revealing examples generated from $S_2$ by CODA, CARROT, and ALERT, respectively. Then, we measured the accuracy of the fine-tuned model on the three evaluation sets to measure its ability of reducing faults.

*2) Results:* Table IV shows the effectiveness of enhancing model robustness with the generated examples by the studied techniques, respectively. The first row (except Column Ori) represents the evaluation set constructed by the corresponding technique, while the second row represents the augmented training set constructed by the corresponding technique. Column Ori lists the accuracy of the fine-tuned model on the original test set. The values in the columns (except Column Ori) represent the ratio of the faults (revealed by the evaluation set) that can be eliminated by the fine-tuned model based on the augmented training set. We found that on most subjects, CODA enhance the model robustness to reduce the largest ratio of faults revealed by CODA, CARROT, ALERT, respectively. On average, the models fine-tuned by CODA can reduce 62.19%, 65.67%, 73.95% of faults revealed by CARROT, ALERT, and CODA respectively, with the improvement of 10.27%, 27.37%, 48.94% over those by CARROT and 33.54%, 11.42%, 27.17% over those by ALERT respectively. Besides, the results of robustness enhancement between different techniques indicate that the examples generated by CODA could subsume those by CARROT and ALERT to a large extent. In five cases, CODA performs worse than ALERT or CARROT, as the augmented training set and the evaluation set generated by the same technique could share a higher degree of similarity (facilitating fine-tuning).

By comparing Column Ori in Table IV and the last column in Table II, the original model and the fine-tuned model via CODA have close accuracy, i.e., all the absolute accuracy differences are less than 1%. The results demonstrate CODA is more helpful to improve model robustness than CARROT and ALERT without damaging the original model performance.

**Answer to RQ2**: CODA helps enhance the model robustness more effectively than CARROT and ALERT, in terms of reducing faults revealed by the examples generated by itself as well as the examples generated by the other two techniques.

### C. RQ3: Contribution of Main Components

*1) Setup:* We studied the contribution of each main component in CODA, i.e., reference inputs selection (RIS), equivalent structure transformations (EST), and identifier renaming transformations (IRT). We constructed four variants of CODA:

- **w/o RIS**: we replaced RIS with the method that randomly selects $N$ inputs from training data as reference inputs.
- **w/o EST**: we removed EST from CODA, i.e., it directly performs identifier renaming transformations after selecting reference inputs.
- **w/o CDG** (code difference guidance in EST): we replaced the code-difference-guided strategy used for EST in CODA with randomly selecting rules for EST.
- **w/o IRT**: we removed IRT from CODA, i.e., it directly checks whether a fault-revealing example is generated after equivalent structure transformations.

*2) Results:* Table V shows the average RFR values of each technique across all the tasks on CodeBERT, GraphCodeBERT, and CodeT5, respectively. The results on each task can be found at our project homepage [19] due to the space limit. CODA outperforms all four variants in terms of average RFR with improvements of 15.79%∼165.27%, demonstrating the contribution of each main component in CODA. Also, reference inputs selection and identifier renaming transformations contribute more than equivalent structure

TABLE V
ABLATION TEST FOR CODA IN TERMS OF AVERAGE RFR

| Model | w/o RIS | w/o EST | w/o CDG | w/o IRT | CODA |
|---|---|---|---|---|---|
| CodeBERT | 30.83% | 62.73% | 63.08% | 35.14% | **73.04%** |
| GraphCodeBERT | 29.49% | 62.41% | 61.98% | 26.24% | **73.62%** |
| CodeT5 | 26.75% | 50.74% | 57.98% | 38.21% | **70.96%** |

transformations. The possible reason is that not all the rules of equivalent structure transformations can be applicable to all the target inputs, but identifier renaming transformations are applicable to all the inputs. We can enrich the rules of equivalent structure transformations in the future to further improve the test effectiveness. The comparison results among CODA, w/o EST, and w/o CDG demonstrate the contribution of our code-difference-guided strategy for applying equivalent structure transformations for testing deep code models. Besides, ALERT targets only identifier-level adversarial example generation, and thus we further compared ALERT with w/o EST for fairer comparison. The results also demonstrate the superiority of the latter, showing the effectiveness of our code difference guided adversarial example generation (despite only considering identifier renaming transformation).

**Answer to RQ3**: All the components of reference input selection, equivalent structure transformations, and identifier renaming transformations make contributions to the overall effectiveness of CODA, demonstrating the necessity of each of them in CODA.

### D. RQ4: Naturalness of Adversarial Examples

*1) Setup:* It is important to check whether the generated fault-revealing examples are natural to human judges [16], [51]. Here, we conducted a user study to compare the naturalness of examples generated by CODA, CARROT, and ALERT, and *our user study shares the same design as the one conducted by the existing work [16]*:

**Data Preparation.** For each subject, we randomly sampled 10 target inputs, and then for each technique on each target input, we randomly sampled a generated example. That is, for each sampled target input, we construct three pairs of code snippets, each of which contains the target input and an adversarial example generated by CODA, CARROT, or ALERT. In total, we obtained 450 pairs of code snippets for the user study due to 15 subjects $\times$ 3 techniques.

**Participants.** Same as the existing work [16], the user study also involves four non-author participants, each of whom has a Bachelor/Master degree in Computer Science with at least five years of programming experience.

**Process.** For objective evaluation, we did not tell participants which technique generates the adversarial example in a pair of code snippets. Also, we highlighted the changes in each pair of code snippets for facilitating manual evaluation. Then, each participant individually evaluated each pair by evaluating to what extent the changes are natural to the code context and the changed identifiers preserve the original semantics,

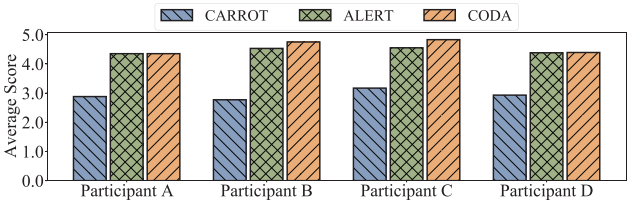



Fig. 5. Average score to evaluate naturalness of examples

TABLE VI
INFLUENCE OF HYPER-PARAMETER $U$.

| U | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|
| CodeBERT | 60.14% | 67.90% | 73.04% | 75.27% | 75.83% |
| GraphCodeBERT | 61.92% | 70.16% | 73.62% | 74.98% | 75.69% |
| CodeT5 | 51.22% | 61.74% | 70.96% | 72.96% | 76.88% |

following the existing work [16]. Specifically, participants gave a score for each pair based on a 5-point Likert scale [52] (1 means strongly disagree and 5 means strongly agree). More details about the design can be found in the existing work [16].

*2) Results:* Figure 5 shows the average score of the examples generated by each technique for each participant. The conclusions from different participants are consistent: the naturalness of the examples generated by CODA and ALERT is closely high (round 4.50 on average), and significantly higher than that by CARROT (just 2.94 on average). ALERT is a naturalness-aware technique, whose core contribution is to ensure naturalness of generated examples, but CODA achieves similar naturalness scores to it, demonstrating that CODA can generate highly natural adversarial examples.

**Answer to RQ4**: The adversarial examples generated by CODA are natural closely to the state-of-the-art naturalness-aware adversarial example generation technique (i.e., ALERT), which is consistently confirmed by participants.

## VI. THREATS TO VALIDITY

The main threat to validity lies in the settings of parameters in CODA. Here, we investigated the influence of two important parameters in CODA (i.e., $U$ and $N$ introduced in Section III-B). They affect the selection of reference inputs. Tables VI and VII show the influence of $U$ and $N$ in terms of average RFR across all the tasks. As $U$ increases, CODA performs better, as incorporating more inputs for the selection based on similarity can increase the possibility of finding more effective reference inputs. Similarly, as $N$ increases within our studied range, more effective ingredients could be included, leading to better effectiveness. However, the amount of increase in terms of average RFR becomes smaller with $U$ and $N$ increasing, and meanwhile incorporating more inputs can incur more costs in similarity calculation or code transformations. Hence, by balancing the effectiveness and efficiency of CODA, we set $U$ to 256 and $N$ to 64 as the default settings in CODA for practical use.

TABLE VII
INFLUENCE OF HYPER-PARAMETER $N$

| N | 1 | 4 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| CodeBERT | 28.08% | 46.33% | 61.07% | 67.12% | 73.04% | 76.38% |
| GraphCodeBERT | 31.84% | 46.46% | 60.40% | 66.12% | 73.62% | 74.93% |
| CodeT5 | 27.05% | 43.71% | 58.19% | 64.82% | 70.96% | 73.25% |

## VII. RELATED WORK

Besides the state-of-the-art techniques compared in our study (i.e., CARROT [13] and ALERT [16]), there are some other adversarial example generation techniques for deep code models [15], [53], [54]. For example, Yefet et al. [55] proposed DAMP, which changes variables in the target input by gradient computation. It only works for the models using one-hot encoding to process code, and thus cannot generate adversarial examples for the models based on state-of-the-art CodeBERT [5], GraphCodeBERT [6], and CodeT5 [18] due to different encoding methods. Zhang et al. [15] proposed MHM, which iteratively performs identifier renaming transformations to generate adversarial examples based on the Metropolis-Hastings [56]–[58] algorithm. MHM underperforms CARROT and ALERT as presented by the existing studies [13], [16]. Pour et al. [59] proposed a search-based technique with an iterative refactoring-based process. Ramakrishnan et al. [40] generated adversarial examples for deep code models via gradient-based optimization, including renaming transformations and dead code insertion. Jha et al. [60] proposed CodeAttack, which adopts the masked language model with greedy search to predict substitutes for vulnerable tokens. All of them do not ensure the naturalness of generated examples, especially with the rule of dead code insertion. Also, these techniques still search for ingredients in the enormous space, limiting their effectiveness. Different from them, our work designs the first code-difference-guided adversarial example generation technique, which can largely reduce ingredient space for improving the test effectiveness.

## VIII. CONCLUSION AND FUTURE WORK

To improve test effectiveness on deep code models, we propose a novel perspective by exploiting the code differences between reference inputs and the target input to guide the generation of adversarial examples. From this perspective, we design CODA, which reduces the ingredient space as the one constituted by structure and identifier differences and designs equivalent structure transformations and identifier renaming transformations to preserve original semantics. We conducted an extensive study on 15 subjects. The results demonstrate that CODA reveals more faults with less time than the state-of-the-art techniques (i.e., CARROT and ALERT), and confirm the capability of enhancing the model robustness.

In the future, we can improve CODA from several aspects. First, CODA successively applies equivalent structure transformations and identifier renaming transformations without backtracking. In the future, we can improve it by backtracking and trying more rounds of transformations if the first round of equivalent structure transformations and identifier renaming transformations fails, which may help transform more target inputs to fault-revealing examples. Second, the number of inputs belonging to the class with the second largest probability may be not enough, even though we did not come across this case in our evaluation. If it occurs in the future, CODA may use the next highest-probability classes as a compromise. Third, we will explore the use of test data to provide reference inputs to further improve CODA. Fourth, we will investigate some other strategies (such as the type obfuscation strategy [61]) to replace the simple strategy using the placeholder <unk> in RIS, in order to further improve the effectiveness of CODA.

## ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China Grant Nos. 62322208, 62002256, 62192731, 62192730, and CCF Young Elite Scientists Sponsorship Program (by CAST),

## REFERENCES

[1] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *31st IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2016, pp. 87–98.

[2] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *33rd International Conference on Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[3] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2018.

[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[5] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[6] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *ICLR*, 2021.

[7] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[8] "Github copilot," https://github.com/features/copilot, Accessed: 2023.

[9] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Artificial intelligence safety and security*. Chapman and Hall/CRC, 2018, pp. 99–112.

[10] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 2021, pp. 397–409.

[11] Q. Shen, J. Chen, J. M. Zhang, H. Wang, S. Liu, and M. Tian, "Natural test generation for precise testing of question answering software," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[12] M. Yan, J. Chen, X. Cao, Z. Wu, Y. Kang, and Z. Wang, "Revisiting deep neural network test coverage from the test effectiveness perspective," *Journal of Software: Evolution and Process*, p. e2561, 2023.

[13] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, "Towards robustness of deep program processing models—detection, estimation, and enhancement," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–40, 2022.

[14] Z. Li, C. Wang, Z. Liu, H. Wang, S. Wang, and C. Gao, "Cctest: Testing and repairing code completion systems," *arXiv preprint arXiv:2208.08289*, 2022.

[15] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1169–1176.

[16] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[17] C. Casalnuovo, E. T. Barr, S. K. Dash, P. Devanbu, and E. Morgan, "A theory of dual channel constraints," in *42nd International Conference on Software Engineering: New Ideas and Emerging Results*. IEEE, 2020, pp. 25–28.

[18] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[19] "Coda," https://github.com/tianzhaotju/CODA, Accessed: 2023.

[20] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[21] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 4159–25.

[22] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *33rd International Conference on Automated Software Engineering*, 2018, pp. 293–304.

[23] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang, "Learning to construct better mutation faults," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[24] T. Gao, J. Chen, Y. Zhao, Y. Zhang, and L. Zhang, "Vectorizing program ingredients for better jvm testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 526–537.

[25] Y. Kang, Z. Wang, H. Zhang, J. Chen, and H. You, "Apirecx: Cross-library api recommendation via pre-trained language model," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 3425–3436.

[26] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1332–1336.

[27] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. Gall, "Adversarial robustness of deep code comment generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–30, 2022.

[28] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *29th International Conference on Tools with Artificial Intelligence*. IEEE, 2017, pp. 45–52.

[29] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *41st International Conference on Software Engineering*. IEEE, 2019, pp. 783–794.

[30] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, "Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1528–1540.

[31] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel, "On the (statistical) detection of adversarial examples," *arXiv preprint arXiv:1702.06280*, 2017.

[32] A. Prakash, N. Moran, S. Garber, A. DiLillo, and J. Storer, "Deflecting adversarial attacks with pixel deflection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8571–8580.

[33] W. Shen, Y. Li, L. Chen, Y. Han, Y. Zhou, and B. Xu, "Multiple-boundary clustering and prioritization to promote neural network retraining," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 410–422.

[34] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 141–151.

[35] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *International Conference on Software Maintenance and Evolution*. IEEE, 2021, pp. 425–436.

[36] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[37] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," *Applied Sciences*, vol. 11, no. 11, p. 4793, 2021.

[38] K. Nakamura and N. Ishiura, "Random testing of c compilers based on test program generation by equivalence transformation," in *Asia Pacific Conference on Circuits and Systems*. IEEE, 2016, pp. 676–679.

[39] H. Cheers, Y. Lin, and S. P. Smith, "Spplagiarise: A tool for generating simulated semantics-preserving plagiarism of java source code," in *10th International conference on software engineering and service science*. IEEE, 2019, pp. 617–622.

[40] J. Henke, G. Ramakrishnan, Z. Wang, A. Albarghouth, S. Jha, and T. Reps, "Semantic robustness of models of source code," in *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2022, pp. 526–537.

[41] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Semi-supervised log-based anomaly detection via probabilistic label estimation," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1448–1460.

[42] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[43] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[44] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, "Source code authorship attribution using long short-term memory based networks," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 65–82.

[45] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[46] "Codechef," https://codechef.com/, Accessed: 2023.

[47] "Tree-sitter," https://tree-sitter.github.io/tree-sitter/, Accessed: 2023.

[48] X. Gao, Y. Feng, Y. Yin, Z. Liu, Z. Chen, and B. Xu, "Adaptive test selection for deep neural networks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 73–85.

[49] H. You, Z. Wang, J. Chen, S. Liu, and S. Li, "Regression fuzzing for deep learning systems," in *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[50] P. Sedgwick, "Spearman's rank correlation coefficient," *Bmj*, vol. 349, 2014.

[51] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *2nd International Conference on Learning Representations, ICLR 2014*, 2014.

[52] A. Joshi, S. Kale, S. Chandel, and D. K. Pal, "Likert scale: Explored and explained," *British journal of applied science & technology*, vol. 7, no. 4, p. 396, 2015.

[53] P. Chen, Z. Li, Y. Wen, and L. Liu, "Generating adversarial source programs using important tokens-based structural transformations," in *26th International Conference on Engineering of Complex Computer Systems*. IEEE, 2022, pp. 173–182.

[54] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U.-M. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," in *International Conference on Learning Representations*, 2021.

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

[56] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[57] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," 1970.

[58] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The American Statistician*, vol. 49, no. 4, pp. 327–335, 1995.

[59] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," in *14th IEEE Conference on Software Testing, Verification and Validation*. IEEE, 2021, pp. 36–46.

[60] A. Jha and C. K. Reddy, "Codeattack: Code-based adversarial attacks for pre-trained programming language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023.

[61] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 243–253.