Variable Renaming-Based Adversarial Test Generation for Code Model: Benchmark and Enhancement

JIN WEN, University of Luxembourg, Luxembourg
QIANG HU*, Tianjin University, China
YUEJUN GUO, Luxembourg Institute of Science and Technology, Luxembourg
MAXIME CORDY, University of Luxembourg, Luxembourg
YVES LE TRAON, University of Luxembourg, Luxembourg

Robustness testing is essential for evaluating deep learning models, particularly under unforeseen circumstances. Adversarial test generation, a fundamental approach in robustness testing, is prevalent in computer vision and natural language processing, and it has gained considerable attention in code tasks recently. The Variable Renaming-Based Adversarial Test Generation (VRTG), which deceives models by altering variable names, is a key focus. VRTG involves substitution construction and variable name searching, but its systematic design remains a challenge due to the empirical nature of these components. This paper introduces the first benchmark to examine the impact of various substitutions and search algorithms on VRTG effectiveness, exploring improvements for existing VRTGs. Our benchmark includes three substitution construction types, six substitution position rank ways and seven search algorithms. Analysis of four code understanding tasks and three pre-trained code models using our benchmark reveals that combining RNNS and Genetic Algorithm with code-based substitution is more effective for VRTG construction. Notably, this method outperforms the advanced black-box variable renaming test generation technique, ALERT, by up to 22.57%.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Computing methodologies** → **Artificial intelligence**.

1 INTRODUCTION

In recent years, deep learning (DL) has achieved remarkable success in automating software engineering tasks, particularly in source code understanding [1]. A variety of deep neural networks (DNNs) with ever better performance have been developed for specific code understanding tasks, such as convolutional neural network for solution classification [31] and recurrent neural network for authorship attribution [2]. Notably, pre-trained models designed for programming languages, e.g., CodeBERT [15], are trending due to their powerful ability to learn source code representations and support a wide range of downstream tasks [29].

Despite their promising results, the reliability of such code models is difficult to guarantee. The reason is that DNNs are generally not robust to data distribution shifts [26] and adversarial examples [37]. Recent studies [21] revealed that the accuracy of code models will drop significantly when facing data with a different distribution from the training data. Other works [44] demonstrated that code models can be easily fooled by a name change

*Corresponding author.

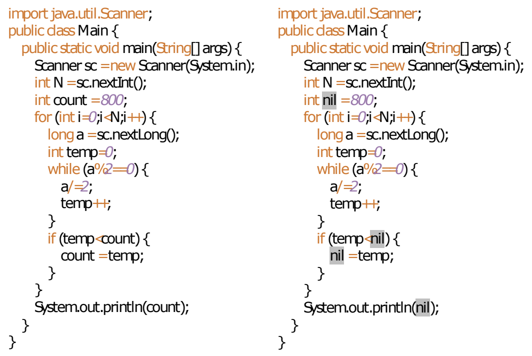
Authors' addresses: Jin Wen, University of Luxembourg, Luxembourg, ruchenshanghai@gmail.com; Qiang Hu*, Tianjin University, China, qianghu0515@gmail.com; Yuejun Guo, Luxembourg Institute of Science and Technology, Luxembourg, yuejun.guo@list.lu; Maxime Cordy, University of Luxembourg, Luxembourg, maxime.cordy@uni.lu; Yves Le Traon, University of Luxembourg, Luxembourg, Yves.LeTraon@uni.lu.

ACM 1557-7392/2025/3-ART
https://doi.org/10.1145/3723353

of variables in the code. Developers often use descriptive variable names to improve code readability, but "meaningful" naming can vary widely across teams and settings [39]. Different organizations tend to adopt different naming conventions, which can substantially influence performance of code models [41]. Because it is impractical to enforce a single naming standard, it is essential to evaluate code models under various naming styles to verify their robustness [41]. Robust models are important because naming choices differ by culture, organization, and domain. If a model lacks robustness, it may perform well only for certain naming conventions, limiting its utility for broader communities. Therefore, measuring model performance across diverse naming styles is crucial for understanding and improving the inclusiveness of automated coding tools. All these facts remind us of the imperative to thoroughly assess the robustness of code models before their integration into real-world applications. Here, robustness refers to the capability of code models to handle unseen data. A higher robustness indicates the model is more practical for usage.

In the field of machine learning (ML), a key method for assessing robustness involves presenting Deep Neural Networks (DNNs) with adversarial test cases and evaluating their performance. An adversarial test case is typically created by introducing subtle perturbations to the original input, which are imperceptible to humans but can cause the DNN to make incorrect predictions. Motivated by the success of adversarial attacks in the fields of computer vision and natural language processing, researchers have developed methods for generating adversarial examples for code models to evaluate their robustness. Among these techniques, black-box variable renaming stands out as the most effective approach for creating adversarial code with minimal changes. While "adversarial testing" often denotes malicious intent, we adopt the term more broadly to refer to systematic stress-testing of code models through subtle yet realistic modifications. In practice, minor edits such as altering identifiers or rearranging statements frequently occur and can stress a model's internal assumptions, much like a traditional adversary. Although there is no malicious actor in daily software development, these small yet pervasive code variations can degrade a model's performance in ways similar to true adversarial scenarios. By generating and evaluating such edits, we assess how reliably code models handle realistic perturbations without relying on idealized or uniform inputs. This approach highlights whether a model maintains consistent performance across natural variations that developers commonly introduce, thereby ensuring robustness in real-world settings. The black-box setting is particularly relevant because large code models, such as Copilot [16], are typically hosted on servers with inaccessible parameters. Generating more such test cases also allows for further adversarial fine-tuning to improve the robustness of code models. Figure 1 illustrates an example of adversarial code generated through variable renaming from an original seed. The code model is used to predict the functionality of these two code samples. By altering just one variable name in the input code (from *count* to *nil*), the code model fails to accurately predict the code.

Existing methods predominantly utilize three components, substitution construction, vulnerable variable localization, and variable name searching to conduct the variable renaming-based test generation (VRTG). Notably, ALERT [44] uses a combination of the Greedy Search and Genetic Algorithm to select optimal variable names from an existing substitution pool for effective adversarial test code generation. Similarly, CodeAttack [24] leverages Greedy Search based on variables' masked probability change, iteratively replacing names within specified similarity and perturbation thresholds. RNNS [48] also determines the most valuable variable name based on Greedy Search. Different from CodeAttack, RNNS sorts variables based on their probability change from a predefined replacement list, rather than solely on masked probability change. Therefore, the development of meaningful substitutions and the selection of an effective search algorithm are critical for a VRTG method. Unfortunately, existing methodologies are often introduced heuristically, with a limited exploration into how various components, such as substitutions and search algorithms (all methods only consider one type of substitution and search algorithm), affect the effectiveness of these methods. There is a need to study the usefulness of VRTG methods with other untouched substitution construction methods and search algorithms.



```java
import java.util.Scanner;
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int N = sc.nextInt();
      int count = 800;
      for (int i=0;i<N;i++) {
         long a = sc.nextLong();
         int temp=0;
         while (a%2==0) {
            a/=2;
            temp++;
         }
         if (temp<count) {
            count = temp;
         }
      }
      System.out.println(count);
   }
}
```

(a) Original code

```java
import java.util.Scanner;
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int N = sc.nextInt();
      int nil = 800;
      for (int i=0;i<N;i++) {
         long a = sc.nextLong();
         int temp=0;
         while (a%2==0) {
            a/=2;
            temp++;
         }
         if (temp<nil) {
            nil = temp;
         }
      }
      System.out.println(nil);
   }
}
```

(b) Variable renamed test code

Fig. 1. Adversarial code generated via variable renaming. Simply swapping out the local variable "count" with "nil" changes the model's prediction of the targeting problem from "Shift only" [4] to "*3 or /2" [5], although the code functionality remains the same.

To fill this gap, this paper introduces a benchmark for variable renaming-based test generation and investigates how to enhance existing methods from three perspectives, 1) substitution construction, 2) vulnerable variable localization, and 3) search algorithm selection. For substitution construction, in addition to collecting substitutions based on the similarity of token embeddings (token-based substitution), we introduce a code-based substitution construction method that considers the similarity between two code embeddings. This ensures that, after variable names are modified, the embeddings of the altered code remain a similar semantic to those of the original code. Besides, we employ substitutions derived from randomly generated variable names as a baseline to check the necessity of meaningful substitution construction. Regarding vulnerable variable localization, we explore how different vulnerable variable localization strategies used in Greedy search-related methodologies affect the effectiveness of VRTG. Finally, regarding search algorithms, our benchmark examines seven distinct methods such as Greedy Search to determine their effectiveness in finding the best variable name change. Additionally, inspired by existing research [44], which indicated that a combination of Greedy Search and Genetic Algorithms can enhance VRTG, our benchmark provides recommendations for the most effective combination of search strategies.

Based on our benchmark, we conducted a comprehensive experimental analysis of adversarial test code generation. Specifically, our study delves into the effectiveness of each test generation method, and assesses how well the generated adversarial code can improve the robustness of code models. We also explore the transferability of these test cases, including their applicability from pre-trained code models to large language models (LLMs) like ChatGPT. Our findings indicate that code-based substitution, which prioritizes substitutions based on the similarity of code embeddings, is particularly effective. When evaluating individual search algorithms, our results show that Genetic Algorithm, when combined with code-based substitution, is the most effective in 7 out of 12 scenarios tested. Additionally, the combination of RNNS and Genetic Algorithm outperforms the combination of Greedy Search and Genetic Search, as previously recommended by ALERT. These results demonstrate that both the method of constructing substitutions and the approach to variable replacement search significantly influence the success of variable renaming test generation. Researchers can enhance the effectiveness of adversarial code generation by optimizing these components based on our benchmark in the future.

To summarize, the main contributions of this paper are:

- We build the first benchmark for generating adversarial tests focused on variable renaming to test the robustness of code models. This benchmark supports diverse methods of substitution construction and variable search algorithms. The benchmark including the source code can be found on our project site [42].
- Our experimental results demonstrate that employing code level embedding similarity for ranking substitutions is a better way to prepare variable name candidates. Additionally, Genetic Algorithm stands out as significantly more effective in searching the variable name to reveal the weakness of code models. The integration of RNNS and Genetic Algorithm with code-based substitutions is the SOTA VRTG method.
- This is the first work that evaluates the transferability of adversarial test cases across different code models, from pre-trained code models to Large Language Models (LLMs), offering novel insights into the robustness of both pre-trained models and LLMs against adversarial tests.

The rest of this paper is organized as follows. Section 2 introduces the important knowledge behind this work. Section 3 introduces why we need a benchmark for adversarial code test generation. Section 4 presents the details of our benchmark. Section 5 explains our experimental design. We then present our empirical results and Section 6. Section 7 provides some practical guidance and discusses the threats that may affect the validity of conclusions. Section 8 reviews related work. Finally, the last section concludes our work.

2 BACKGROUND

2.1 Deep Learning for Code

The application of deep learning models to address software tasks has emerged as a prominent research area, particularly in the field of program understanding, which has attracted significant attention. Generally, using deep neural networks to learn program information can be roughly divided into three steps, 1) code representation, 2) code embedding, and 3) downstream task learning. Code representation tends to transform raw code data to machine-readable digit values. There are two widely used code representation techniques, the first one is to treat raw code data as natural language and represent each token in order with indices. The second way is to represent code based on its structure or run-time information, e.g., abstract syntax tree or data flow. After code representation, the program has been transformed to data with discrete input space which cannot be learned by the model directly. Therefore, it is necessary to use embedding layers to encode the represented data to float-format vectors with continuous input space, named code embedding. Finally, these embedding vectors can be used to learn different downstream tasks, for example, function naming.

More recently, to further improve the performance of programming language models, researchers follow the experience from the field of natural language processing to use pre-trained models for code learning. The basic idea is to utilize multiple language data (including both programming language and natural language) to train an embedding model (step 2 mentioned above) and then use this model for downstream tasks. Some advanced powerful pre-trained models have been proposed, e.g., CodeBERT [15], GraphCodeBERT [19] and CodeT5 [40], which are studied in this work.

2.2 Robustness of Deep Learning Model

Robustness testing is one way to measure the generalization ability of deep learning models. There are two types of model robustness, adversarial robustness and natural robustness. For adversarial robustness testing, the practical way is to utilize adversarial test generation methods to generate adversarial code test cases, and then test the model using such examples. For simplicity, we will use adversarial code test cases and adversarial examples interchangeably in this work. Where adversarial examples are clean test data with some manually injected human imperceptible perturbation. Generally, a model with higher adversarial robustness is harder to be attacked. Natural robustness means the ability of the model to correctly predict the unseen natural test data. A widely used method to measure the natural robustness is to use data with different data distribution (both in-distribution and out-of-distribution) to test the model. In this work, we focus on generating adversarial examples to measure the robustness of code models. Meanwhile, we also explore how our generated adversarial example can be used to enhance both adversarial and natural robustness.

To enhance the robustness of models, the most effective way is adversarial fine-tuning. Adversarial fine-tuning contains two steps, firstly it generates adversarial examples from the training data against the pre-trained model. Then, it trains the pre-trained model with additional few epochs using generated examples to force the model to learn both the clean data and perturbation information. In our work, we conduct experiments to show that the adversarial examples generated by our method are more useful in improving the model robustness according to adversarial fine-tuning.

3 PROBLEM DEFINITION AND MOTIVATION

3.1 Adversarial Test Case Generation

Let $\mathcal{P}$ and $\mathcal{Y}$ represent the input and output space of the code model $M : \mathcal{P} \rightarrow \mathcal{Y}$. In our work, we mainly focus on code classification tasks, thus, $\mathcal{Y}$ represents the label space. Following the previous work [13], a code data is defined as a 3-tuple $P = (C, se, sy)$ and $P \in \mathcal{P}$, where C represents all the context in the code; se and sy are the semantic and syntax of the code. The se refers to the semantic information of the code, capturing its meaning or functionality. The sy refers to the syntactic structure of the code, which includes its formal arrangement and adherence to programming language rules. This distinction between semantics and syntax is crucial for tasks like data augmentation and code understanding with semantic-persevering and syntax-breaking transformations [13].

*Definition 3.1 (**Adversarial Test Case**).* Given a code P, an adversarial attack method $\mathsf{Attack} : \mathcal{P} \rightarrow \mathcal{P}'$, an adversarial test case is generated by $P' = \mathsf{Attack}(P)$, that $P'.C \neq P.C \wedge P'.se = P.se \wedge P'.sy = P.sy \wedge M(P') \neq M(P)$.

The goal of adversarial test generation is to prepare as much as possible adversarial test cases to reveal the buggy behavior of code models. In this study, we consider the widely studied attack method – Variable Renaming-Based Adversarial Test Generation (VRTG) for adversarial test case generation, which is defined as follows:

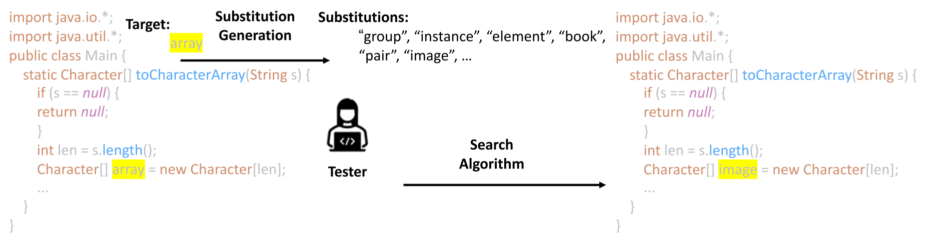
*Definition 3.2 (**Variable Renaming-Based Adversarial Test Generation (VRTG)**).* Given a code P, a list of variable names $Var = (var_1, var_2, ..., var_n) \wedge Var \in P.C$, where n is the number Var. VRTG is to generate code that $P'.Var = VRTG(P.Var) \wedge \exists (P'.var_i \neq P.var_i) \wedge M(P') \neq M(P)$, where $1 \leq i \leq n$.

3.2 Importance of Robustness Testing

Assessing the robustness of deep learning models is one goal in machine learning testing [49]. Robustness indicates the generalization of DNN models, which means the ability of DNN models to handle data with different data distributions to the original training data. A deep learning model with high robustness is important for its practical usage. However, existing works [21, 45] demonstrated that code models are not robust, and suffer from data distribution shift. Thus, how to reveal the shifted distribution and fix this problem is an urgent task. Adversarial test generation is the representative technique in robustness testing used to generate distribution-shifted data. Similar to the test generation techniques, e.g., fuzz, in the conventional software engineering field, adversarial test generation methods tend to generate tests to reveal the faults in DNN models accordingly. To guarantee the reliable usage of code models, it is necessary to study the adversarial test generation problem in code models.

3.3 Limitation of Existing Works

Figure 2 illustrates the common workflow of the variable renaming adversarial code generation. It involves two main steps, substitution construction and variable name search.



```
import java.io.*;
import java.util.*;
public class Main {
  static Character[] toCharacterArray(String s) {
    if (s == null) {
    return null;
    }
    int len = s.length();
    Character[] array = new Character[len];
    ...
  }
}
```

```
import java.io.*;
import java.util.*;
public class Main {
  static Character[] toCharacterArray(String s) {
    if (s == null) {
    return null;
    }
    int len = s.length();
    Character[] image = new Character[len];
    ...
  }
}
```

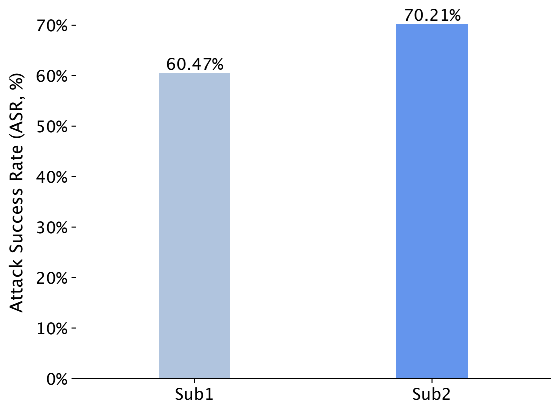
Fig. 2. Workflow of variable renaming test generation.

- **Substitution construction.** It is essential that the candidate name list of the given variable name closely resembles the original name to ensure the changes remain imperceptible in the generated test case. This step addresses the primary challenge of generating appropriate substitutions. In existing studies, such as ALERT[44], the original variable's position is masked first, and a generative model (the masked version of CodeBERT [15]) is employed to suggest potential names for the masked position.
- **Vulnerable variable localization.** A code sample could contain multiple variables. The order in which variable names are replaced one by one can significantly affect the efficiency of VRTG, particularly when using methods related to Greedy Search. Current techniques involve different position ranking strategies to help localize vulnerable variables. For example, ALERT ranks variable names based on the magnitude of their probability change when masking them [44]. RNNS calculates ranking reference weights based on the probability changes resulting from replacing original variables with names from a predefined candidate set [48].
- **Variable name search.** Once substitutions are ready and the variable is localized, the next step is to select the most suitable variable name from the candidate list to create a new code. Attempting every possible code variation is the most straightforward approach, but impractical due to the larger search space. Therefore, an efficient search method to figure out variable replacement is critical to guide the whole search process. This could include employing strategies like the Metropolis-Hastings Modifier algorithm (MHM), Greedy Search, and others to streamline the selection process.

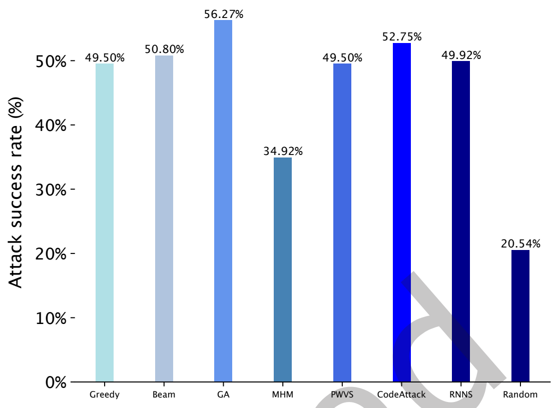
Even though each component mentioned above contributes to the adversarial code generation, we noticed that existing works only considered simple combinations of these two components, i.e., only explored one type of substitution and one search algorithm. This raises questions – what are the best ways to build substitution and find the vulnerable variable? Which search algorithm is more effective in finding the best replacement path? More importantly, what is the best combination for a more powerful adversarial test generation? We give one example of adversarial code generation under different settings as shown in Figure 3. We can see that the state-of-the-art method, ALERT, has significant performance improvement (from 60.47% to 70.21%) by simply changing its substitution. Moreover, different search algorithms also highly affect the attack success rate with a performance gap of 9.74%. There is, therefore, a need to conduct a comprehensive study to explore the best way to generate adversarial code and answer the aforementioned questions.

4 BENCHMARK

We build the first benchmark to support variable renaming-based adversarial test generation for code models, covering three substitution construction methods, three vulnerable variable localization methods, and seven variable name replacement search algorithms.



(a) ASR under different substitutions. Method–ALERT.



(b) ASR under different search algorithms

Fig. 3. Attack success rate(ASR) under different configurations. Model: CodeBERT. Task: Code classification.

4.1 Substitution Construction Method

Substitutions consist of alternative variable names that can be used to replace a designated variable. It is crucial for these substitutions to be natural, as the goal of adversarial test generation is to introduce subtle, imperceptible perturbations into the target code. Our benchmark includes three substitution construction methods, token-based substitution, originally proposed by ALERT, and a newly code-based substitution. In addition, we introduce a random substitution method as a general baseline, which is neglected by existing works.

- **Token-based substitution** uses a pre-trained masked language model (CodeBERT masked language model) to generate all possible substitutions first, and then computes the similarity between the generated tokens and the original variable name. After that, the tokens with high similarity will be put in the substitution list for further adversarial test generation.
- **Code-based substitution** follows the same step as the token-based substitution to generate all possible variable names. The difference is this method computes the similarity based on the whole code embeddings instead of only token-wise embeddings.
- **Random substitution** directly generates random character strings one by one as the new variable name.

4.2 Vulnerable Variable Localization Method

The ranking of the original variables targeted for replacement (also called word saliency) highly affects the efficiency of adversarial test generation. However, the effectiveness of this mechanism has not been thoroughly investigated. Therefore, our benchmark includes various word saliency settings to facilitate detailed studies. These settings include:

- **Mask with unknown** As mentioned above, PWWS proposes to set a word to unknown (out of vocabulary), and calculate its change in output probability to determine the ranking order of all original variables [32].
- **Predefined set replacement** Contrary to the masking with unknown in PWWS and ALERT, RNNS adopts a predefined set replacement strategy to calculate changes and rank all original variables [48].
- **Static v.s. Dynamic** Existing algorithms often utilize a static order, which is established before any replacements are made. However, the actual importance of variables might change during the replacement

process. Therefore, we propose to recalculate the importance of variables after each replacement, which is called dynamic ranking.

4.3 Search Algorithm

Search algorithm here refers to methods that find the variable replacement to trigger the vulnerable behavior of code models. Thus, one search algorithm can be one test generation method. Our benchmark supports seven singular search algorithms including six (MHM, PWVS, Greedy Search, Genetic Algorithm, CodeAttack and RNNS) that have been studied by existing works, and one new (Beam Search) and more combinations of two-step search methods.

- **Metropolis-Hastings Modifier (MHM)**, uses Metropolis-Hastings sampling [47] to find the best variable name for adversarial test. Specifically, a Markov chain Monte Carlo sampling approach is the core search algorithm in MHM.
- **Greedy Search** is a fundamental approach widely employed in optimization problems. The central tenet of Greedy Search in variable replacement here lies in picking out the best available substitution at each step, which is guided solely by immediate gains of classification probability change in the current step.
- **PWVS** is the variable-based version of its original **P**robability **W**eighted **W**ord **S**aliency substitution algorithm (PWWS) [32]. It uses the classification probability difference of different word/variable to determine the substitution selection and replacement order. The difference between PWVS and Greedy Search is that PWVS recalculates the variable's probability change after each replacement.
- **Genetic Algorithm (GA)**, inspired by the principles of natural selection and genetics, performs genetic operators to generate new substitution solutions to get adversarial code [44]. GA here represents variable and substitution pairs as genes and builds chromosomes on the pairs. Then, GA performs genetic operators, including selection, crossover, and mutation, to generate new chromosomes. During the process, GA proposes the fitness function to evaluate the quality of chromosomes, and select the best chromosomes that can mislead the victim model.
- **Beam Search** is the modified version of Greedy Search, which employs a selective approach to maintain only a limited set (beam width) of the most promising solution replacement operation at each step. Compared with Greedy Search, Beam Search can improve part of local optimal substitutions with a higher query count.
- **CodeAttack** mainly adopts Greedy Search, and changes its gain comparison with threshold limit to find the best variable name for adversarial test [24]. During each iteration, CodeAttack limits the similarity change and perturbation change before and after replacement to keep its consistency with the original code snippet.
- **RNNS** constructs a large original substitute set to assess the importance of each variable name, then reuses the importance with Greedy Search to generate the final adversarial test code [48].

The state-of-the-art adversarial code generation method, ALERT [44], leverages pre-trained models to generate natural substitution first, and applies a combination of two-step search methods (Greedy Search and Genetic Algorithm) to generate adversarial codes. We will also discuss how to combine different search algorithms in Section 6.4

5 EMPIRICAL STUDY

Based on our benchmark, we conduct an empirical study to investigate the effectiveness of existing VRTG methods with different settings and explore the best setting (i.e., the best combination of components).

5.1 Research Question

Our analysis focuses on answering the following research questions.

RQ1: *How do different components affect the effectiveness and efficiency of adversarial code generation?*

- **RQ1.1:** *How do different search algorithms affect the performance of VRTG?* Given a fixed group of substitutions, we want to know which search algorithm is more effective in finding the best variable name to reveal the faults of code models.
- **RQ1.2:** *How do different substitutions affect the performance of VRTG?* Better substitutions contain more valuable variable names that are useful for adversarial test generation. We plan to figure out a good way to construct substitutions.
- **RQ1.3:** *How do different word saliency settings affect the performance of VRTG?* Since a program snippet can contain multiple variables, we tend to study if the order of variables to be replaced has a significant impact on the adversarial test generation.

RQ2: *What is the best combination of two-step search algorithms?* Following existing work [44], we study how to combine different search algorithms to further boost the effectiveness of adversarial code generation.

RQ3: *What is the effectiveness of adversarial test cases generated by different methods for model robustness enhancement?* In addition to studying how can adversarial test generation reveal faults in code models, we also plan to check the effectiveness of using generated code to harden code models' robustness.

RQ4: *What is the transferability of adversarial test cases across different models?* Transferability measures how general the generated adversarial test cases are. It is an important property computed by the ASR of adversarial examples generated from the based model to other models not used in the generation process. A higher transferability indicates the adversarial test case is a more common fault and can be used to test different models to save generation time. Nowadays, transferability has become more important for the robustness evaluation of many large models due to the limitations of query count, query cost and inaccessible probability, e.g., ChatGPT. Therefore, this research question plans to study the transferability of adversarial test cases.

5.2 Dataset and Model

Table 1. Statistics of Dataset. GraphCodeBERT model on Clone Detection uses label count=1, and other models on Clone Detection use label count=2.

Task	Train/Valid/Test	Labels	Language	Accuracy		
				CodeBERT	GraphCodeBERT	CodeT5
Author Attribution	528/–/132	66	Python	82.58%	75.00%	78.03%
Clone Detection	90,102/4,000/4,000	1, 2	Java	93.68%	96.28%	95.55%
Code Classification	62,500/-/12500	250	Java	95.25%	97.07%	92.07%
Vulnerability Detection	21,854/2,732/2,732	1	C	63.98%	62.45%	62.74%

In this work, we consider four downstream code understanding tasks and three pre-trained code models as shown in Table 1.

- **Authorship attribution.** By inferring the characteristics of programmers by their written source code, the model can determine the author of a given code snippet. This task is important to provide credit for a programmer's contribution and is useful for plagiarism detection. We use the dataset provided by [44].
- **Clone detection** This task checks if two codes are semantically equivalent or not, which can prevent bug propagation and facilitate software maintenance. We use the widely used BigCloneBench benchmark dataset [36].
- **Code classification.** Code (solution) classification Given a source code file, the model predicts the class of its targeting problem, e.g., to print heights of the top three mountains in descending order. We use the Java250 dataset provided by the CodeNet project [31]. This dataset includes source code files from two online judge websites, AIZU Online Judge and AtCoder.

- **Vulnerability detection.** This task focuses on identifying vulnerable functions, which is crucial to protect software systems. We use the Devign dataset, which is manually collected by [50].

As shown in Table 1, we developed three SOTA code models: CodeBERT, GraphCodeBERT, and CodeT5 for each task.

- **CodeBERT.** CodeBERT is a BERT-like, pre-trained model focusing on comprehending and generating code [15]. It shows powerful performance for code understanding, completion, and generation.
- **GraphCodeBERT.** It is also a pre-trained model tailored for code. Different from CodeBERT, GraphCodeBERT is pre-trained by using the combination of token information and graph information of code snippets. Here, the graph information is extracted from the data flow of the code. In this way, GraphCodeBERT can enhance the comprehension of complex code structures and relationships [19].
- **CodeT5.** CodeT5 is a more recent and powerful pre-trained model based on the T5 backbone structure. It leverages the capabilities of T5 to excel in code understanding and generation tasks and showcases its prowess in natural language understanding and code synthesis [40].

Besides, in the evaluation of adversarial test case transferability, we conduct experiments on ChatGPT[1]. ChatGPT is a state-of-the-art language processing AI model, and it provides context-aware responses across diverse domains. In software engineering, ChatGPT embodies a significant advancement in employing NLP models for code-related tasks.

5.3 Configuration

All experiments are conducted on a 2.6 GHz Intel Xeon Gold 6132 CPU with an NVIDIA Tesla V100 16G SXM2 GPU. The experiments consist of three parts, adversarial test generation, adversarial fine-tuning, and transferability. We use the following configuration for each part.

- **Adversarial test generation configuration.** When generating substitutions, we produce a maximum of 60 candidates for each variable at the token level and 500 at the code level, keeping the top 30 for token and 90 for code based on their similarity to the original embedding. For Beam Search, we use a beam width of 2 in the following experiments. For the Genetic Algorithm, we follow the settings of ALERT with a child_size of 64, dynamic maximal iterations, and a crossover rate of 0.7. For MHM, the maximum iteration is set to 100, in line with ALERT. For CodeAttack, the similarity threshold is set to 0.5 and the perturbation threshold is set to 0.1 as suggested by the authors. The human study of adversarial variable naming is conducted among three proficient developers with more than 5 years of experience in software development.
- **Adversarial fine-tuning configuration** During the adversarial fine-tuning, we divide the test data previously used in adversarial test generation into two equal parts. One part is used for adversarial fine-tuning, while the other is separately reserved for subsequent final evaluation. Successful adversarial codes are mixed with the original codes, and the model is retrained individually for each adversarial test generation method. The retraining adopts the Adam optimizer, configured with an epsilon of 1e-8 and a learning rate of 5e-05. The model is retrained across four tasks: Author Attribution (50 epochs), Clone Detection (2 epochs), Code Classification (8 epochs), and Vulnerability Detection (15 epochs). We also launch the adversarial test generation on the retrained models to evaluate the effectiveness of such adversarial fine-tuning.
- **Transferability configuration** To assess the transferability of generated code across code models, we use successful adversarial codes generated using various adversarial test generation methods on the original model to launch adversarial tests on other un-retrained models across all four tasks. For evaluating transferability to ChatGPT, ten successful adversarial codes are randomly sampled from the Authorship Attribution task for each adversarial test generation method while targeting CodeBERT/GraphCodeBERT/CodeT5. The evaluation

[1]https://openai.com/blog/chatgpt

focuses on their capability to mislead ChatGPT's author prediction, providing a measure of their transferability to LLMs. We fill the adversarial codes into the designed templates, as shown in Figure 4, then save the prediction response from ChatGPT.

```
I will give you two code blocks separated by --.
You need to give me the prediction whether the two code blocks are from the same author.
code1:
int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *pic)

{

    av_assert0(0);


}
---
code2:
int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *IXSfZ)

{

    av_assert0(0);


}
---
Please direct give me the result as True or False.
```

Fig. 4. ChatGPT prompt input example

5.4 Evaluation Methods

We measure the effectiveness, efficiency, and transferability of adversarial test generation methods using attack success rate (ASR), query count, and transfer ASR as well as Absolute transfer ASR, respectively.

- **Attack success rate (ASR).** We assess the proportion of adversarial examples that are successfully generated. We measure the percentage of successfully created adversarial examples. Note that adversarial test generation is applied solely to test samples that are correctly predicted on trained models. A high ASR indicates effective adversarial test code generation and a lack of robustness in the target model.
- **Query count.** We record the number of queries used to generate successful adversarial code to assess the efficiency of each VRTG method. A lower query count number indicates that the VRTG method needs less effort the find the proper test cases.
- **Transfer ASR.** It quantifies the percentage of adversarial examples crafted based on one code model that can successfully deceive another different code model. Increased transferability highlights the ability of the adversarial test method to create more complex and challenging adversarial examples. This demonstrates the wide-reaching effectiveness and potential for generalization of the adversarial test case across various models.
- **Absolute transfer ASR.** Contrary to the transfer ASR, this ASR is computed by dividing the number of successfully transferred codes by the number of correctly classified original test data. This metric provides a direct representation, expressed in percentage (%), of the absolute number of successfully transferred codes.

Table 2. ASR (normal font and black color) and query count (text in italic and brown) on models across four tasks. Values highlighted in blue indicate the best ASR.

Task	ASR(%), *Query Count(avg)*															
	Greedy		Beam		GA		MHM		PWVS		CodeAttack		RNNS		Random	
CodeBERT																
Author	33.94	*46172.48*	33.94	*86839.45*	**38.53**	*619612.84*	23.85	*224206.42*	33.94	*56467.89*	35.78	*337149.54*	33.03	*45840.37*	23.85	*1453.21*
Clone	23.67	*30486.02*	23.83	*55589.51*	**25.43**	*125936.75*	22.71	*224738.96*	23.67	*35565.65*	**25.43**	*150569.02*	23.46	*30458.05*	2.61	*879.3*
Classification	49.5	*21580.97*	50.8	*36201.81*	**56.27**	*125412.92*	34.92	*210451.12*	49.5	*24492.86*	52.75	*87041.05*	49.92	*21025.68*	20.54	*649.43*
Vulnerability	54.58	*17796.34*	**54.92**	*27949.43*	41.08	*26286.78*	38.84	*183042.22*	54.63	*20519.68*	53.78	*66988.96*	54.46	*17436.61*	17.11	*566.13*
Average	40.42	*29008.95*	40.87	*51645.05*	40.33	*224312.32*	30.08	*210609.68*	40.44	*34261.52*	**41.94**	*160437.14*	40.22	*28690.18*	16.03	*887.02*
GraphCodeBERT																
Author	48.48	*42176.77*	48.48	*76021.21*	**65.66**	*386118.18*	28.28	*218730.3*	48.48	*51829.29*	45.45	*300200.0*	47.47	*41543.43*	34.34	*1385.86*
Clone	17.5	*27509.01*	20.96	*57325.27*	21.24	*141812.05*	14.8	*246017.29*	20.33	*36489.54*	**23.41**	*149976.61*	20.57	*31195.2*	2.49	*892.34*
Classification	37.76	*23517.58*	39.31	*40580.72*	**48.72**	*105783.39*	25.57	*230217.9*	37.76	*26648.7*	39.8	*102294.16*	37.98	*23169.26*	15.02	*688.0*
Vulnerability	58.02	*16217.76*	59.35	*24781.84*	**75.9**	*67609.35*	70.57	*96236.58*	60.68	*17551.16*	56.69	*55008.11*	62.06	*15130.97*	31.07	*499.59*
Average	40.44	*27355.28*	42.02	*49677.26*	**52.88**	*175330.74*	34.81	*197800.52*	41.81	*33129.67*	41.34	*151869.72*	42.02	*27759.72*	20.73	*866.45*
CodeT5																
Author	73.79	*28308.74*	73.79	*48585.44*	74.76	*170803.88*	58.25	*149883.5*	72.82	*35301.94*	**76.7**	*182139.81*	71.84	*28260.19*	50.49	*1169.9*
Clone	4.29	*17061.25*	4.26	*31989.09*	3.79	*25911.41*	5.34	*128202.09*	**8.16**	*39145.24*	5.57	*84602.49*	8.14	*33716.59*	1.94	*895.58*
Classification	69.19	*16883.48*	71.2	*25954.84*	62.97	*46281.01*	56.25	*143745.45*	**72.11**	*20031.95*	70.94	*62216.27*	68.82	*16760.8*	61.08	*495.16*
Vulnerability	70.97	*11287.14*	72.35	*15206.18*	**77.01**	*20896.5*	76.55	*74159.45*	71.46	*12807.29*	66.1	*36948.71*	71.32	*11093.29*	53.73	*318.49*
Average	54.56	*18385.15*	55.4	*30433.89*	54.63	*65973.2*	49.1	*123997.62*	**56.14**	*26821.6*	54.83	*91476.82*	55.03	*22457.72*	41.81	*719.78*
Average	45.14	*24916.46*	46.1	*43918.73*	**49.28**	*155205.42*	38.0	*177469.27*	46.13	*31404.27*	46.03	*134594.56*	45.76	*26302.54*	26.19	*824.42*

6 RESULTS ANALYSIS

6.1 RQ1.1: Impact of Search Algorithm

First, we study how different search algorithms affect the performance of code generation methods. We use Attack Success Rate (ASR) and Query Count to measure the effectiveness and efficiency of each method, respectively. Table 2 presents the results. Regarding the effectiveness (ASR) of search algorithms, no single algorithm consistently outperforms the others across all tasks and models. However, the Genetic Algorithm and CodeAttack perform relatively better than other methods, and often achieve the highest average ASR across various tasks and models. Notably, the Genetic Algorithm achieves an ASR of 77.01% on the CodeT5 model for the vulnerability detection task, indicating the non-robustness of code models to variable renaming tests. Most search algorithms can achieve nearly double the ASR compared to the worst results from Random Search, with MHM performing the worst in most situations except for Random Search. Considering the efficiency (query count) of each search algorithm, Greedy Search and RNNS stand out, having approximately half the query count of the best ASR method, the Genetic Algorithm. Interestingly, Greedy Search, on average, saves 6.2 times the query count compared to the Genetic Algorithm while still being effective in generating adversarial code. The results of ASR and query count help balance the effectiveness and efficiency of search algorithms based on different scenario requirements.

Answer to RQ1.1: Genetic Algorithm is the most effective method in terms of generating adversarial code tests. Considering the trade-off between effectiveness and efficiency, Greedy Search is still recommended for variable renaming-based adversarial generation.

6.2 RQ1.2: Impact of Substitution

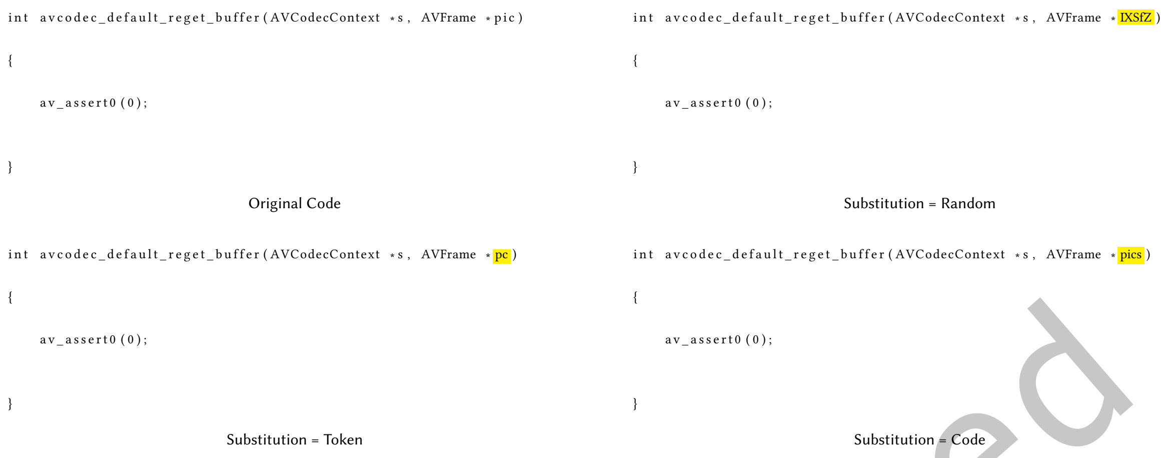
Table 3. Average ASR and query count on models(CodeBERT, GraphCodeBERT and CodeT5) across four tasks using different substitutions. Values highlighted in blue indicate the best across different substitutions.

Task	ASR (%), *Query Count(avg)*															
	Greedy		Beam		GA		MHM		PWVS		CodeAttack		RNNS		Random	
Random																
Author	50.61	*5945.89*	52.23	*10811.76*	**63.59**	*346233.53*	50.36	*44861.95*	50.61	*18221.24*	17.11	*24389.91*	51.26	*5725.01*	36.23	*1336.32*
Clone	3.72	*7323.81*	3.84	*13495.63*	**5.64**	*91040.52*	3.73	*107303.06*	4.29	*15663.08*	3.52	*24783.57*	4.36	*8772.71*	2.35	*889.07*
Classification	58.98	*5917.9*	61.27	*9432.45*	**72.54**	*58740.2*	59.91	*76964.89*	60.28	*9044.54*	41.7	*13183.88*	58.75	*5874.45*	32.21	*610.86*
Vulnerability	69.41	*5261.59*	71.04	*7323.87*	**71.12**	*18608.93*	68.41	*66708.22*	69.51	*7656.71*	31.92	*3646.5*	68.12	*5263.0*	33.97	*461.41*
Average	45.68	*6112.3*	47.09	*10265.93*	**53.22**	*128655.79*	45.6	*73959.53*	46.17	*12646.39*	23.56	*16500.97*	45.62	*6408.79*	26.19	*824.42*
Token																
Author	52.07	*38885.99*	52.07	*70482.03*	**59.65**	*392178.3*	36.8	*197606.74*	51.75	*47866.37*	52.64	*273163.12*	50.78	*38548.0*	-	-
Clone	15.16	*25018.76*	16.35	*48301.29*	16.82	*97886.74*	14.28	*199652.78*	17.39	*37066.81*	**18.14**	*128382.7*	17.39	*31789.94*	-	-
Classification	52.15	*20660.68*	53.77	*34245.79*	**55.99**	*92492.44*	38.92	*194804.82*	53.12	*23724.5*	54.5	*83850.5*	52.24	*20318.58*	-	-
Vulnerability	61.19	*15100.41*	62.21	*22645.82*	**64.66**	*38264.21*	61.99	*117812.75*	62.26	*16959.38*	58.86	*52981.92*	62.62	*14553.62*	-	-
Average	45.14	*24916.46*	46.1	*43918.73*	**49.28**	*155205.42*	38.0	*177469.27*	46.13	*31404.27*	46.03	*134594.56*	45.76	*26302.54*	-	-
Code																
Author	62.83	*50125.25*	64.75	*88366.68*	**68.34**	*338796.02*	38.0	*181909.6*	63.74	*57762.61*	60.03	*373951.35*	64.81	*48205.94*	-	-
Clone	11.77	*35269.63*	12.42	*68756.33*	13.46	*130674.83*	7.96	*202449.05*	13.11	*50216.11*	**13.95**	*174343.54*	13.06	*44771.5*	-	-
Classification	63.19	*34179.4*	**65.0**	*54401.39*	63.72	*96603.14*	36.73	*200759.13*	64.12	*37296.61*	64.44	*134899.0*	62.72	*34088.45*	-	-
Vulnerability	66.14	*17838.89*	67.09	*26042.53*	**68.57**	*41843.65*	65.95	*97550.94*	66.26	*19800.81*	63.05	*58730.62*	66.22	*17595.25*	-	-
Average	50.98	*34353.29*	52.32	*59391.73*	**53.52**	*151979.41*	37.16	*170667.18*	51.81	*41269.03*	50.37	*185481.13*	51.7	*36165.29*	-	-

Next, we study how different substitution construction methods influence adversarial code generation. Table 3 presents the results of each test generation method using different substitution construction methods, including a baseline generated by random character selection. Figure 5 illustrates generated codes from these three substitution methods, and the replaced variables are highlighted in yellow in the code. As we can see from Figure 5, the use of variable names affects the naturalness of code. It is necessary to measure the impact of different substitutions on the naturalness of generated code. To do so, we conduct a human study on the Authorship Attribution task. Specifically, we ask developers to give a score (chosen from 0, 1, and 2) of each generated code where 0 indicates the code contains unnatural variable names, 1 indicates it is difficult to judge, and 2 indicates the code is natural. After that, we normalize the score to a range of 0% to 100%. The results of the human study are shown in Table 4.

Table 4. Human study about variable renaming naturalness on Authorship Attribution with sample N = 3 from each search algorithm. The higher the detection rate, the more likely the variable renaming is recognized as abnormal.

Substitution	**Human Detection Rate (%)**		
	Human 1	Human 2	Human 3
Random	97.22%	97.22%	100.0%
Token	9.72%	7.64%	6.25%
Code	34.72%	20.83%	27.78%



```
int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *pic)
{
    av_assert0(0);

}
```
Original Code

```
int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *IXSfZ)
{
    av_assert0(0);

}
```
Substitution = Random

```
int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *pc)
{
    av_assert0(0);

}
```
Substitution = Token

```
int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *pics)
{
    av_assert0(0);

}
```
Substitution = Code

Fig. 5. Three adversarial codes different in substitution generated by Greedy Search on vulnerability detection.

From the results, the first conclusion is that substitution methods significantly affect the performance of adversarial test generation. The ASR gap can be as large as 42.92% (between Author-CodeAttack with Random and Code substitutions). Random substitution shows notable ASR improvement, achieving up to 72.54% on Classification-GA. However, Random substitution is easily detected, with over 90% abnormal detection rate by human inspection. Figure 5 shows code examples generated by using different substitutions. The variable name generated by random substitution, *IXSfZ*, is obviously uncommon. While variable names generated using token-based substitution (*pc*) and code-based substitution (*pics*) are semantically similar to the original name (*pic*). Comparing Code and Token substitutions, our proposed code-based substitution outperforms the token-based substitution in all cases except for the clone detection task. This result suggests that code models rely more on character-level information than code meaning for detecting code clones than other tasks. Consistent with results in RQ1, the Genetic Algorithm performs well with code-based substitution, achieving the highest ASR on most tasks.

In terms of efficiency, the query count results show that token-based substitution is more efficient than code-based substitution. However, the difference is negligible, with the maximum gap being only 1.38 times (Clone detection tasks – Greedy search). This indicates that substitution construction methods have a limited impact on the efficiency of adversarial test generation. Overall, code-based substitution is a relatively better method for VRTG.

Answer to RQ1.2: Substitution highly affects the effectiveness of VRTG methods. Our proposed code-based substitution is relatively better than token-based substitution in terms of improving the ASR and keeping similar efficiency. On the other hand, token-based substitution is powerful in generating code with the highest naturalness.

6.3 RQ1.3: Impact of Word Saliency

The word saliency is used to localize the vulnerable variable name to be changed. Normally, word saliency methods rank variable names based on their vulnerability, a higher vulnerability indicates after changing this variable name, the model is easier to produce a wrong prediction. Word saliency is previously considered in

Table 5. Average ASR and query count of Beam Search on CodeBERT, GraphCodeBERT and CodeT5 with different word saliency settings. The best results are highlighted in blue.

Task	**Beam Search - ASR (%),** *Query Count(avg)*									
	Predefined Mask		Predefined Replacement		Dynamic Mask		Dynamic Replacement		Random	
					Token					
Author	52.07	*70482.03*	**52.09**	*69418.6*	51.75	*86339.18*	**52.09**	*294495.19*	51.45	*82808.32*
Clone	16.35	*48301.29*	17.64	*58704.55*	**17.7**	*68112.92*	17.59	*188513.97*	16.88	*59558.0*
Classification	53.77	*34245.79*	53.92	*33619.34*	54.79	*39154.96*	**55.04**	*94484.51*	53.24	*39551.05*
Vulnerability	62.21	*22645.82*	62.56	*22251.57*	62.45	*25821.59*	**64.86**	*68610.0*	61.94	*23540.73*
Average	46.1	*43918.73*	46.55	*45998.52*	46.67	*54857.16*	**47.39**	*161525.92*	45.88	*51364.52*
					Code					
Author	64.75	*88366.68*	64.71	*85837.06*	**65.06**	*101713.33*	64.71	*278002.19*	60.85	*112561.04*
Clone	12.42	*68756.33*	**13.55**	*83094.97*	13.32	*92930.7*	13.49	*216738.83*	12.95	*83733.33*
Classification	65.0	*54401.39*	64.64	*54205.98*	**65.96**	*59259.17*	65.86	*112405.5*	65.0	*63966.48*
Vulnerability	67.09	*26042.53*	67.02	*25894.45*	67.15	*29142.41*	**69.71**	*69864.96*	66.82	*27405.98*
Average	52.32	*59391.73*	52.48	*62258.12*	52.87	*70761.4*	**53.44**	*169252.87*	51.4	*71916.71*

Greedy Search algorithms, which can be also employed in Beam Search, PWVS, and RNNS. In this part, we analyze whether the word saliency mechanism contributes to adversarial test generation. Table 5 summarizes the results of methods with different word saliency calculation strategies. The results demonstrated that, generally, dynamic mask or dynamic replacement outperforms predefined mask or predefined replacement in terms of average ASR. Considering the efficiency, dynamic methods require higher query counts compared to predefined methods. Random word saliency does not significantly affect ASR and does not reduce the query count compared to other methods. Overall, the maximum average ASR gap among different word saliency methods is only 2.04% (dynamic replacement against random). The predefined mask method significantly saves the query count, costing three times fewer query counts than the dynamic replacement method.

Answer to RQ1.3: There is a trade-off between the effectiveness and efficiency of VRTG methods when considering word saliency techniques. VRTG methods with dynamic word saliency achieve higher ASRs, with an average improvement of 1% over predefined word saliency methods. However, predefined word saliency methods are more efficient, making them three times faster than those with dynamic word saliency.

Overall, the above three experiments demonstrated that the search algorithm, substitution, and word saliency highly affect the effectiveness and efficiency of variable renaming-based adversarial code test generation.

Finding: Among all configurations, the Genetic Algorithm with code-based substitution is the best adversarial test generation method for code models that has the highest ASR and affordable query cost.

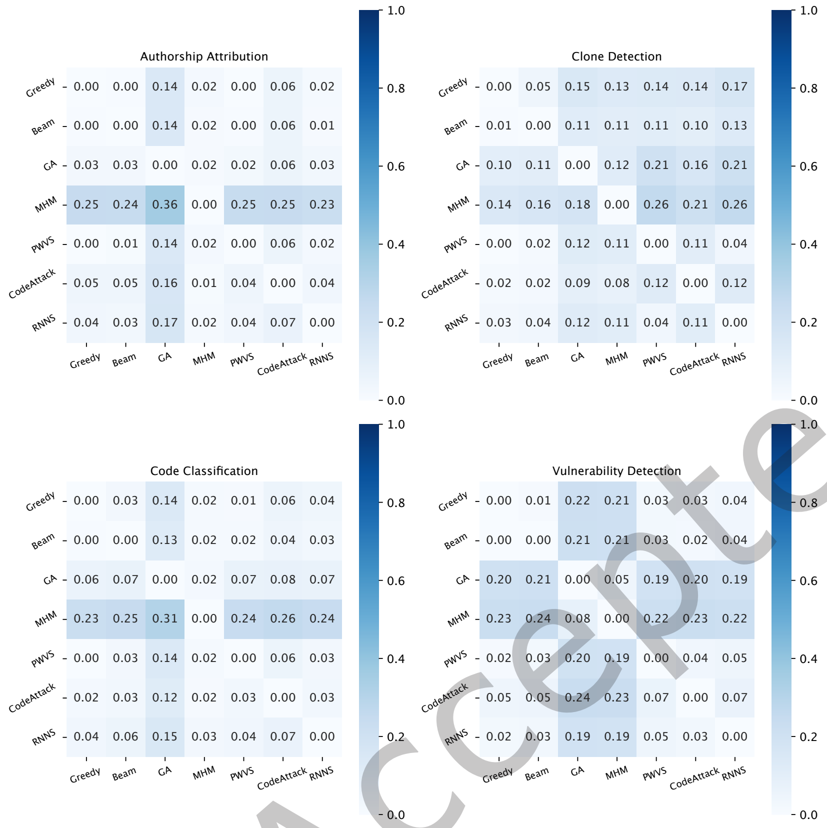


Fig. 6. An example of search algorithms' non-overlapping adversarial test case distribution of Table 2. A higher score indicates the generated test cases by these two methods are much more different.

6.4 RQ2: Search Algorithm Combination

Existing works [44] have considered combining two search algorithms (Greedy Search and Genetic Algorithm) to improve adversarial test generation. However, only one type of combination has been studied. It is worth exploring further enhancing adversarial code generation through search algorithm combinations.

First, we need to check if different search algorithms are *independent*. Here, *independent* means that one generation method can generate adversarial code from inputs that another method cannot. Unlike ALERT's code reuse mechanism during different steps of the search process, our combination explores individual search without overlapping code. Under this situation, the combination of search algorithms is meaningful. Figure 6 illustrates the average non-overlapping distribution of different search algorithms on various code tasks using CodeBERT, GraphCodeBERT, and CodeT5. The detailed results are available on our project site [42]. The average results can be generalized to all models. A higher non-overlapping score indicates that two search algorithms can find more different test cases. Therefore, this two search algorithm could be a better combination to generate more useful test cases. Notably, MHM generates more unique adversarial test cases than other search algorithms, as shown by the darker cells in Figure 6.

Table 6. Average ASR(%) of search algorithm combination on CodeBERT, GraphCodeBERT and CodeT5. The best results are highlighted in blue. Substitution(Sub in Table) = Token and GA + Greedy indicates ALERT(denoted by *). Red downside arrow indicates the ASR drop compared to the base method, while the blue upside arrow indicates the ASR improvement.

				Token					
Task		Beam	PWVS	CodeAttack	GA	RNNS		Greedy	Beam
Author		53.04(↑)	53.05(↑)	53.3(↑)	60.61(↑)	52.07(↑)		55.95(↑)	55.95(↑)
Clone	**MHM**	18.58(↑)	19.69(↑)	19.75(↑)	19.22(↑)	19.65(↑)	**CodeAttack**	18.59(↑)	18.71(↑)
Classification	**+X(%)**	54.81(↑)	54.38(↑)	55.52(↑)	57.67(↑)	54.38(↑)	**+X(%)**	55.86(↑)	56.59(↑)
Vulnerability		81.63(↑)	80.07(↑)	81.07(↑)	68.26(↑)	80.19(↑)		63.32(↑)	63.71(↑)
Average		52.02(↑)	51.8(↑)	**52.41(↑)**	51.44(↑)	51.57(↑)		48.43(↑)	48.74(↑)
Task		Greedy*	Beam	PWVS	CodeAttack	RNNS		PWVS	RNNS
Author		52.07(↓)	61.6(↑)	61.28(↑)	**63.16(↑)**	61.92(↑)		55.63(↑)	55.29(↑)
Clone	**GA**	16.1(↓)	19.32(↑)	20.5(↑)	**20.65(↑)**	20.63(↑)	**CodeAttack**	19.92(↑)	19.93(↑)
Classification	**+X(%)**	52.15(↓)	61.21(↑)	**61.45(↑)**	61.35(↑)	61.34(↑)	**+X(%)**	56.83(↑)	56.8(↑)
Vulnerability		70.8(↑)	**82.29(↑)**	80.65(↑)	81.65(↑)	80.77(↑)		65.21(↑)	65.45(↑)
Average		47.78(↓)	56.1(↑)	55.97(↑)	**56.7(↑)**	56.17(↑)		49.4(↑)	49.37(↑)
				Code					
Task		Beam	PWVS	CodeAttack	GA	RNNS		Greedy	Beam
Author		65.06(↑)	64.05(↑)	60.67(↑)	68.95(↑)	65.92(↑)		67.19(↑)	66.3(↑)
Clone	**MHM**	13.71(↑)	14.48(↑)	14.91(↑)	14.49(↑)	14.48(↑)	**CodeAttack**	14.44(↑)	14.56(↑)
Classification	**+X(%)**	65.98(↑)	65.31(↑)	65.59(↑)	65.68(↑)	64.84(↑)	**+X(%)**	67.7(↑)	68.47(↑)
Vulnerability		**85.17(↑)**	84.68(↑)	83.88(↑)	75.67(↑)	84.45(↑)		68.21(↑)	68.8(↑)
Average		**57.48(↑)**	57.13(↑)	56.27(↑)	56.2(↑)	57.42(↑)		54.39(↑)	54.53(↑)
Task		Greedy*	Beam	PWVS	CodeAttack	RNNS		PWVS	RNNS
Author		63.44(↓)	71.28(↑)	71.25(↑)	70.28(↑)	**74.64(↑)**		66.27(↑)	69.34(↑)
Clone	**GA**	12.23(↓)	15.66(↑)	16.49(↑)	16.47(↑)	**16.49(↑)**	**CodeAttack**	15.37(↑)	15.42(↑)
Classification	**+X(%)**	63.19(↓)	70.96(↑)	**71.14(↑)**	70.6(↑)	70.77(↑)	**+X(%)**	68.64(↑)	68.5(↑)
Vulnerability		74.65(↑)	84.47(↑)	84.02(↑)	83.41(↑)	83.73(↑)		68.19(↑)	68.37(↑)
Average		53.38(↓)	60.59(↑)	60.73(↑)	60.19(↑)	**61.41(↑)**		54.62(↑)	55.41(↑)

Next, we combine two search algorithms for adversarial code generation. Table 6 presents the ASR of methods, where GA+Greedy* under token-based substitution refers to ALERT [44]. Since ALERT reuses the replaced temporary code during the two-step search process, it can steadily outperform Greedy Search but may not always beat Genetic Algorithm. Among these results, the highest ASR of 85.17% is found in MHM+Beam with code substitution on the vulnerability task. The ASR on the most challenging task, clone detection, is 20.65%, indicating the potential of search algorithm combinations. From the average ASR results, we see that RNNS and Genetic Algorithm with code substitution on non-clone tasks is the best combination, achieving the highest average ASR in most situations. Actually, Genetic Algorithm combined with PWVS, CodeAttack, and RNNS also shows comparable top performance in most cases. Notably, on average, compared to ALERT, GA+RNNS has 8.39% and 8.03 higher ASRs with code substitution and token substitution, respectively. Additionally, code substitution is still better than token substitution on non-clone detection tasks, which is consistent with the conclusion in RQ1.2.

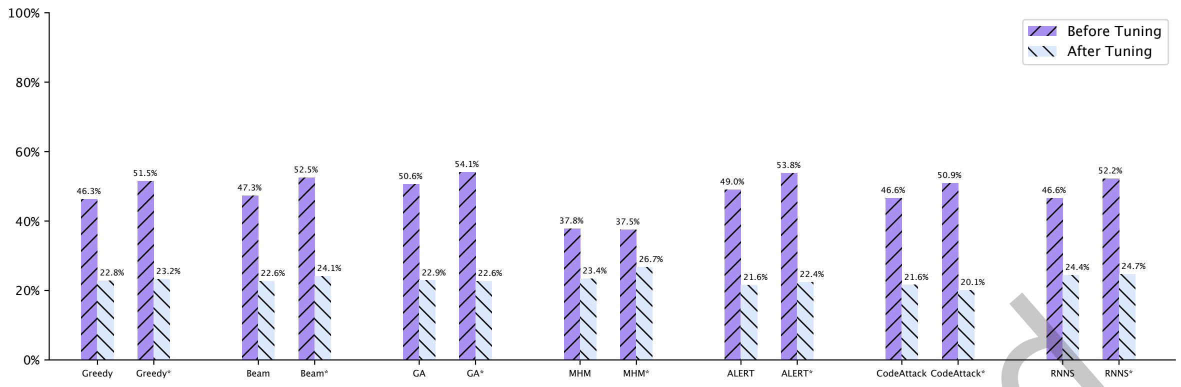


Fig. 7. Average ASR on Model=CodeBERT, GraphCodeBERT and CodeT5 before and after adversarial tuning. Method with * denotes code-based substitution.

Answer to RQ2: Combining two different search algorithms is an effective way to boost adversarial code generation. The combination of RNNS and Genetic Algorithm is a better way that achieves 8.39% higher ASR on average than combining Greedy Search and Genetic Algorithm suggested by existing work ALERT.

6.5 RQ3: Robustness Enhancement

Like conventional software testing, the final goal of finding bugs is to fix these bugs. We study the effectiveness of using the generated adversarial code (as patches) to fix the *bugs* in code models – to enhance the robustness of code models. Table 7 presents the accuracy and ASR of code models after adversarial fine-tuning of CodeBERT models and Figure 7 illustrates the average results before and after adversarial fine-tuning. The whole results can be found on our project site [42].

Our results demonstrate that adversarial fine-tuning significantly improves the accuracy and robustness of code models. Fine-tuned models exhibit a modest accuracy improvement, with increases ranging from 2.33% to 22.22%. Moreover, robustness enhancements are more substantial, varying from 10.77% to 31.4%. Specifically, code classification shows the most significant benefits from fine-tuning, while clone detection proves to be the most resistant to robustness enhancements. When comparing different types of substitutions, adversarial codes created with code-based substitution result in a greater robustness improvement of 26.94%, compared to 23.56% for token-based substitution. Despite these advancements, the fine-tuned models still encounter challenges, maintaining an average Attack Success Rate (ASR) of 37.09% after fine-tuning, indicating that adversarial test generation remains a potent threat. Previous works [8] also showed that simply using generated test cases to enhance code models leads limited robustness improvement. The reason is conjectured to be the discrete input space is hard to learn for code models. Testers can always find variable names that code models did not learn well. This phenomenon highlights the ongoing need for further research into methods to enhance the robustness of code models effectively. Additionally, the Greedy Search method on vulnerability detection with token substitution leads to a higher ASR post-fine-tuning, suggesting that retraining under such conditions could inadvertently increase the model's vulnerability. Furthermore, models fine-tuned using Random Search show the worst improvement in robustness. Not only adversarial codes from Random Search have a higher ASR on retrained models, but also the ASR increases further when the retrained models are targeted by the search algorithm.

These findings emphasize the critical importance of maintaining consistent variable naming not only for improving code readability but also for safeguarding machine learning models against sophisticated adversarial attacks. The observed decrease in performance due to adversarial retraining underscores the efficacy of variable renaming testing in revealing latent sensitivities within models. This study serves as a call to action for continued exploration into robust machine learning methodologies.

Table 7. Average absolute accuracy (ΔAcc) and ASR (ΔASR) changes after fine-tuning on CodeBERT, GraphCodeBERT and CodeT5. The adversarial test cases used to evaluate ΔACC and ΔASR is generated from original trained models and the data will not be used in fine-tuning. The ASR* denotes the ASR on final retrained model after tuning with separate adversarial test cases (no overlapping with test data). Arrows with up (down) direction indicate the performance increased (decreased) compared with previous values. The larger ASR decline indicates the better robustness enhancement. The bigger accuracy increase indicates the better performance enhancement.

Token												
Task	**Greedy(%)**			**Beam(%)**			**GA(%)**			**MHM(%)**		
	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*
Authorship	**22.22**↑	27.76↓	47.42↓	**22.22**↑	27.21↓	47.04↓	21.88↑	**39.51**↓	**56.13**↓	**22.22**↑	5.07↓	33.78↓
Clone	**3.8**↑	8.69↓	8.44↓	3.51↑	10.83↓	6.96↓	3.19↑	7.85↓	**14.55**↓	3.39↑	8.55↓	6.71↓
Classification	3.05↑	45.54↓	37.8↓	**3.19**↑	46.79↓	36.46↓	3.17↑	46.59↓	40.02↓	2.51↑	29.34↓	28.01↓
Vulnerability	15.82↑	12.33↓	62.43↑	15.74↑	13.74↓	**65.22**↑	16.13↑	16.83↓	60.87↓	16.6↑	14.82↓	43.03↓
Average	**11.22**↑	23.58↓	39.02↓	11.16↑	24.64↓	38.92↓	11.09↑	**27.69**↓	**42.9**↓	11.18↑	14.44↓	27.88↓
Task	**ALERT(%)**			**CodeAttack(%)**			**RNNS(%)**			**Random(%)**		
	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*
Authorship	**22.22**↑	26.53↓	49.8↓	**22.22**↑	28.78↓	48.68↓	**22.22**↑	18.38↓	48.34↓	**22.22**↑	-1.89↑	32.6↓
Clone	3.33↑	9.81↓	9.4↓	3.55↑	11.14↓	11.67↓	3.17↑	**13.09**↓	7.28↓	3.17↑	-21.92↑	2.59↑
Classification	3.03↑	45.64↓	35.66↓	2.84↑	**47.52**↓	**41.91**↓	2.99↑	44.75↓	36.03↓	2.81↑	17.68↓	19.64↓
Vulnerability	17.41↑	**27.73**↓	60.58↓	14.03↑	12.45↓	46.14↓	**17.54**↑	12.49↓	48.15↓	14.47↑	-17.95↑	31.13↓
Average	**11.5**↑	**27.43**↓	**38.86**↓	10.66↑	24.97↓	37.1↓	11.48↑	22.18↓	34.95↓	10.67↑	-6.02↑	21.49↓
Code												
Task	**Greedy(%)**			**Beam(%)**			**GA(%)**			**MHM(%)**		
	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*
Authorship	**22.22**↑	30.24↓	59.72↓	**22.22**↑	34.32↓	**61.41**↓	**22.22**↑	**42.66**↓	59.04↓	**22.22**↑	-2.75↑	39.2↑
Clone	3.57↑	0.92↓	**11.38**↓	3.22↑	-0.85↑	7.86↓	2.83↑	3.32↓	10.05↓	3.31↑	-0.45↑	5.58↓
Classification	3.14↑	55.94↓	38.26↓	2.89↑	**57.34**↓	39.94↓	3.02↑	53.1↓	44.22↓	2.46↑	24.94↓	30.11↓
Vulnerability	**19.46**↑	26.13↓	63.26↓	15.19↑	22.63↓	**65.78**↓	16.93↑	26.67↓	59.49↓	16.54↑	21.32↓	49.89↓
Average	**12.1**↑	28.31↓	43.16↓	10.88↑	28.36↓	**43.75**↓	11.25↑	**31.44**↓	43.2↓	11.13↑	10.77↓	31.19↓
Task	**ALERT(%)**			**CodeAttack(%)**			**RNNS(%)**			**Random(%)**		
	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*	ΔAcc	ΔASR	ASR*
Authorship	**22.22**↑	32.54↓	56.53↓	**22.22**↑	41.02↓	57.03↓	**22.22**↑	32.38↓	59.63↓	**22.22**↑	-1.89↑	32.6↓
Clone	3.06↑	2.97↓	11.01↓	**3.69**↑	**8.3**↓	9.99↓	3.5↑	5.24↓	9.05↓	3.17↑	-21.92↑	2.59↑
Classification	**3.14**↑	55.58↓	38.47↓	2.91↑	56.74↓	**44.8**↓	2.33↑	52.94↓	38.86↓	2.81↑	17.68↓	19.64↓
Vulnerability	18.2↑	**34.49**↓	54.33↓	16.65↑	17.12↓	52.88↓	16.55↑	19.62↓	65.69↓	14.47↑	-17.95↑	31.13↓
Average	**11.66**↑	**31.4**↓	40.09↓	11.37↑	30.79↓	41.18↓	11.15↑	27.54↓	**43.31**↓	10.67↑	-6.02↑	21.49↓

Answer to RQ3: Adversarial fine-tuning modestly improves the accuracy of code models by 11.36% and significantly boosts their robustness by 26.94%. However, in some instances, it may reduce robustness further. Code-based substitution methods outperform token-based ones in enhancing robustness. Random Search techniques, on the other hand, struggle to improve robustness effectively. Despite these enhancements, fine-tuned models continue to be vulnerable to adversarial test cases, exhibiting an average ASR of 37.09% post-fine-tuning. This summary underscores the complex impact of different fine-tuning strategies on model performance.

6.6 RQ4: Transferability

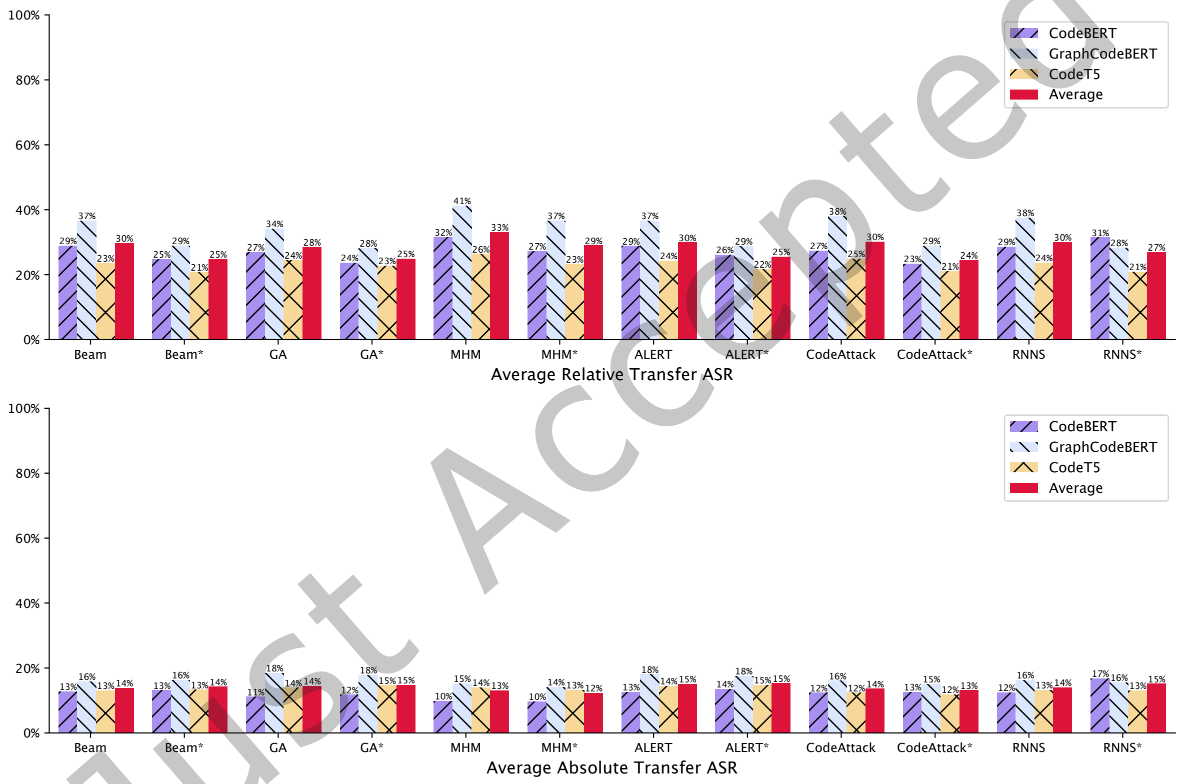


Fig. 8. Average transfer ASR of search algorithms among all tasks. Method name with * denote code-based substitution, other names denote token-based substitution.

Finally, we evaluate the transferability of generated adversarial code. Figure 8 summarizes the average transfer ASR of each method across all considered datasets. Besides, we also consider the absolute transferability of adversarial code calculated by the percentage of the number of successfully transferred code divided by the number of original test data as shown in the second row in Figure 8. From the results, the first conclusion we can draw is that MHM achieves the best results with 33% transfer ASR. Compared to the results in Table 2 and Table 6, we can say that even though MHM is less effective than other methods, it can generate more *general*

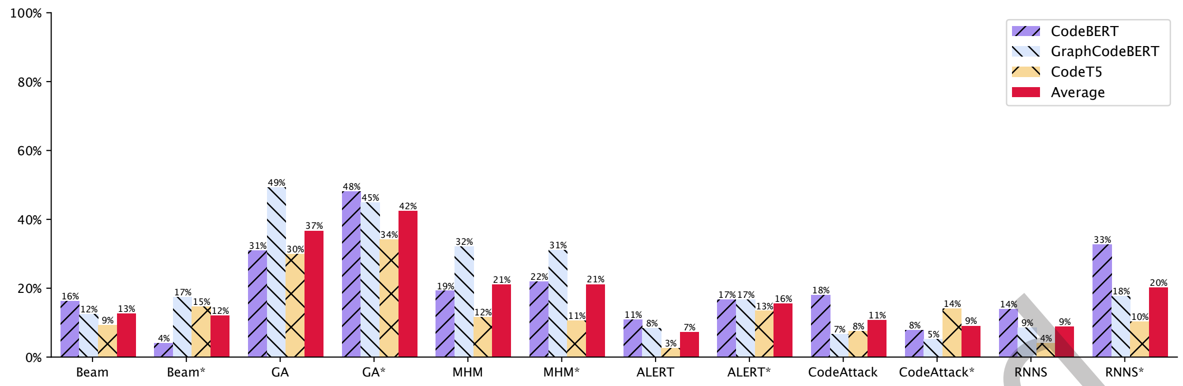


Fig. 9. Relative Transfer ASR of Task = Authorship Attribution on ChatGPT (GPT-3.5). Method name with * denote code-based substitution, other names denote token-based substitution.

adversarial codes (i.e., test cases generated by MHM are more likely common bugs of all code models). However, considering the absolute transferability, we found that all methods are at a similar level, with around 15% absolute transferability on average. Transfer attack for code models is a challenging task. Besides, we found that the choice of substitutions has a limited impact on the transfer ASR. Using the token-based substitution is slightly better than using code-based substitution in this scenario with a maximum relative transfer ASR gap of 6%.

In addition to studying the pre-trained code models, in this part, we also consider evaluating the transferability of generated adversarial code on a large language model, ChatPGT (GPT-3.5). The reason is that it is more practical to evaluate the robustness of models we can query a limited number of times. Figure 9 depicts the results of the ChatGPT study. The first conclusion we can draw is that ChatPGT is not robust and many faults can be easily revealed by using adversarial codes generated by other models. Then, comparing different adversarial code generation methods, we can see that the transferability of adversarial codes generated by MHM (the one that performs the best in pre-trained code models) drops a lot. Instead, Genetic Algorithm with code-based substitution is the best method in terms of transferability with a maximum average transfer ASR of 42%.

Answer to RQ4: Considering pre-trained code models only, MHM generates more *general* adversarial codes that can be easily transferred from one code model to another. The construction of substitution does not highly affect the transferability of generated adversarial codes. Considering LLMs, ChatGPT is not a robust code model where faults can be easily revealed (with a 42% transfer ASR on average) by using adversarial codes generated by pre-trained code models.

7 DISCUSSION

7.1 Guidance

- As revealed by our study (see RQ1.2 and Table 3), code models show significant drops in prediction accuracy when variable names change. For instance, examined code models exhibit an Attack Success Rate (ASR) of up to 72.54% under random renaming (Classification task), indicating high sensitivity to naming variations. Therefore, in practical settings, we recommend feeding code with consistent, meaningful variable names to minimize these reliability issues.

- For developers who plan to build custom models, our results (see RQ3, Table 7, and Figure 7) indicate that testing with standard test data alone is insufficient. Although the original models achieve high accuracy on in-distribution data, adversarial test generation reveals many "hidden" vulnerabilities. We also show that adversarial fine-tuning improves robustness by an average of 26.94% in ASR reduction, confirming that additional adversarial training data is crucial before deployment.
- For researchers aiming to advance robustness testing of code models, our results (see RQ1.2 and RQ2) show that clone detection remains the hardest task to fool using variable renaming (*e.g.*, Table 3 and Table 6 shows Clone tasks typically have the lowest ASR, often below 21%). This suggests that more sophisticated transformations, beyond simple renaming, are needed to alter perceived code similarity. Moreover, while most existing Variable Renaming-Based Test Generation (VRTG) methods target classification tasks, expanding these adversarial approaches to other code tasks such as code generation or refactoring may uncover broader vulnerabilities that remain unaddressed in current literature.
- When testing closed-source models like ChatGPT, our RQ4 experiments show that adversarial test cases generated from GraphCodeBERT or CodeT5 can still compromise ChatGPT with a transfer ASR of up to 49% (see Figure 9). Notably, MHM produced more generalizable adversarial examples for pre-trained models. However, Genetic Algorithm combined with code-based substitution transferred slightly better to ChatGPT. Hence, we recommend first generating adversarial inputs from open-source models with multiple search algorithms (including MHM or GA) to uncover potential security or reliability gaps in large-scale, closed-source systems.

7.2 Threats to Validity

The internal threat is the implementation of the benchmark methods. All our attack methods are modified from projects provided by the original papers [24, 44, 48], and the implementation of code models is from the famous open-source project Hugging Face [23].

The external threat lies in our considered search algorithms, substitution construction methods, code tasks, and code models. We include all the search algorithms that appeared in existing code attack works [44, 47] and introduce one stronger search method Beam Search in our benchmark. For the substitution construction method, we considered the state-of-the-art method, token-based substitution, in our benchmark and proposed a new one, code-based substitution. The experimental results show that our code-based substitution is better than token-based substitution. For the used code tasks and code models, we follow the recent work [44] and add all their used tasks and models to our benchmark. Besides, we also consider a more powerful code model, CodeT5, in our benchmark.

The construct threat can be the configuration of attack and fine-tuning. For the adversarial attack, we use all the settings that are recommended by the original papers in our study. For the code model fine-tuning, we follow the work [20] that provides suggestions to configure the retraining process to conduct our study.

8 RELATED WORK

We review the related works from three perspectives, deep learning for source code, adversarial attacks in CV and NLP, and adversarial attacks in source code.

8.1 Deep Learning for Source Code

Deep neural networks have been utilized to help developers in multiple source code tasks in the past decade [29], such as code search [18] and code clone detection [43], where source code is processed as text or graph data by code representation techniques. Nowadays, there are two main types of code representation methods, sequence-based representation, and graph-based representation. For sequence-based representation, the raw code is seen as text data and represented as a sequence of tokens, and then fed to the model. By using this technique, CodeBERT, TFix [6], CoTexT [30] achieved impressive success in different downstream tasks. For the graph-based code representation, the raw code is transformed into different types of structural data first, e.g., AST and control flow. Then, structural data can be fed to the model directly or processed by the graph neural networks and then fed to the model. The most popular graph representation-based technique is GraphCodeBERT [19], which learns code information from its data flow and handles downstream tasks. CodeT5, another notable model in the realm of vector representation learning, advances the field of software engineering by fostering enhanced code understanding [40]. The model's advanced design and architecture allow it to effectively learn representations from large-scale multilingual code-related datasets, ensuring extensive applicability and utility in diverse programming contexts and environments.

Contrary to the prevailing focus on code learning, our research meticulously examines the robustness of various code representations against adversarial attacks. This investigation involves the generation and application of adversarial examples targeting diverse coding models, offering keen insights into their resilience and vulnerabilities. Our research, expanding beyond traditional boundaries, applies adversarial examples from external model attacks to other models and Large Language Models like ChatGPT. This approach ensures an exhaustive and nuanced understanding of code representation robustness within adversarial contexts. Further, we carry out adversarial fine-tuning and reattacking to highlight the robustness disparity across different code models. This emphasizes our dedication to detailed and comprehensive robustness assessment.

8.2 Machine Learning Testing

Testing machine learning systems involves complex processes not typically encountered in traditional software testing. A primary challenge is the absence of explicit programming logic in ML systems, where their behaviors are derived from data [33]. This characteristic complicates predicting outputs from given inputs, making the creation of conventional test cases challenging. Moreover, the inherent non-determinism of many ML systems, especially those using algorithms that involve randomness such as stochastic gradient descent, adds another layer of complexity [11]. Ensuring consistent outcomes across different runs, even with the same input, necessitates robust testing methodologies. Extensive reviews, such as the one conducted by Zhang *et al.*, explore various methods developed to assess ML workflows, components, and properties. These studies provide valuable insights into the evolving landscape of ML testing [49]. A focal area within ML testing is the generation of adversarial examples, categorized in the test generation process [49]. These generated inputs specifically designed to challenge the robustness of ML models by attempting to deceive them, highlighting potential vulnerabilities [33]. Additionally, with the growing integration of AI and software engineering, innovative methodologies such as metamorphic testing are gaining traction [38]. Metamorphic testing involves systematically altering inputs and observing whether the outputs maintain expected patterns, which is a useful approach when traditional test oracles are unfeasible. These developments underscore the importance of advanced testing strategies in enhancing the reliability and integrity of ML systems.

Our work focuses on test generation for code-targeted machine learning models. We are the first to build the benchmark to study the impact of different components on the effectiveness of variable renaming-based adversarial test generation methods.

8.3 Adversarial Attacks in CV and NLP

Depending on the accessibility of DL models, adversarial attacks in CV and NLP are categorized into three groups, white-box, gray-box, and black-box. In the white-box setting, attacks have full access to the parameters and model architectures. Thus, gradient information is always used in white-box attacks. For image data in CV tasks, there are the fast gradient sign method (FGSM) [17], basic iterative method (BIM, also called i-FGSM) [27], projected gradient descent (PGD) [28], and C&W [10] attacks. For text data in NLP tasks, different from images where the input space is continuous, the text-based attacks are more challenging and usually need to create a vocabulary in advance and then search for the best fit. The gradient information is used in the searching step, such as the HotFilp [14] and the attacking to training (A2T) [46] attacks. Gray-box attacks only have access to the model architectures. In contrast, black-box attacks can only rely on the model's prediction. Some attacks are also done by transferring adversarial examples from a substitute model [7]. For image data, the square attack simply utilizes the predicted probabilities or logits (score-based) [3]. The boundary attack [9, 34] and fast adaptive boundary [12] attack calculate the perturbation by querying the predicted class (decision-based). For text data in NLP tasks, well-known attacks include TFLexAttack[22], TextFooler [25] and probability-weighted word saliency (PWWS) [32].

Different from existing works that focus on image or text data, our work targets code data, which is the key component in software systems.

8.4 Adversarial Attacks in Source Code

Yefet *et al.* [45] proposed the white-box attack, discrete adversarial manipulation of programs (DAMP), that changes the name of local variables or adds dead code to mislead DL models. As for variable renaming, originally, the variable name is represented as a one-hot vector. DAMP utilizes the gradient information to calculate the perturbation that is added to the one-hot vector. Since the input space of the source code is discrete, the generated vector is no longer one-hot. Thus, DAMP selects the most likely name via Argmax. As for dead-code insertion, DAMP adds a new variable declaration in the code. However, this attack is easy to defend by cleaning the code before deployment. Similarly, Srikant *et al.* [35] proposed the projected gradient descent-based joint optimization attack based on the gradient information. In a black-box manner, Zhang *et al.* [47] proposed the Metropolis-Hastings modifier (MHM) algorithm that applies the MH sampling to search for the substitute variable name. However, Yang *et al.* [44] noticed that MHM does not necessarily generate natural adversarial codes according to human judgments. To this end, they developed the naturalness aware attack, ALERT, to explicitly consider the natural semantics of generated code. Specifically, ALERT uses the masked language prediction function of pre-trained models to create the vocabulary for each local variable. Instead of using pre-trained models for the substitutes, Jha and Reddy [24] proposed the CodeAttack that utilizes the WordPiece algorithm for tokenization and find the appropriate substitute tokens. Additionally, CodeAttack relies on not only variables to perform the adversarial attack but also keywords, class names, method names, arguments, and operators. Compared to existing code attacks that focus on variable renaming, our work has a major difference. Almost all previous attacks utilize a disordered vocabulary where all substitutes share equal importance for the targeting variable, which will potentially increase the query time if the vocabulary size is very large.

Diverging from prior research, our work embarks on a journey to probe the distinctions in code models subjected to varied attack methods. Employing a unified experimental framework allows us to unveil the attack success rate and efficiency across diverse attack methods more effectively. Moreover, we endeavor to perform a thorough experimental analysis of pivotal steps such as substitution generation and word saliency in existing methods across disparate scales' code model. This analysis elucidates the unique features and suitable applications of various methods. In conclusion, our research confirms the transferability and effectiveness of prevailing attack methods against models of diverse scales, including Large Language Models, thereby enhancing the comprehensiveness of our study in this critical domain.

9 CONCLUSION

We developed the first benchmark for studying variable renaming-based adversarial test generation to evaluate the robustness of code models. Our benchmark incorporates seven search algorithms, three substitution construction methods, and five word saliency calculation strategies, offering a comprehensive framework for robustness testing. Using our benchmark, we systematically investigated the generation effectiveness of different adversarial test generation methods, their ability to enhance the robustness of code models, and the transferability of their generated code. Our findings highlight that while random substitution is highly effective in generating adversarial code, the resulting code is often unnatural and fails to deceive human reviewers. In contrast, our proposed code-based substitution approach outperforms the token-based substitution which is recommended by the SOTA attack method ALERT in clone detection tasks. Among single-step search algorithms, Beam Search with code-based substitution and dynamic replacement word saliency performs the best for generating adversarial test codes. For two-step search algorithm, combining RNNS and Genetic Algorithm with code-based substitution outperforms the ALERT-recommended combination of Greedy Search and Genetic Algorithm with token-based substitutions. Besides, although adversarial fine-tuning is useful for enhancing the robustness of code models, our study shows that these adversarial test generation methods remain effective even against the enhanced models. Moreover, our results reveal that ChatGPT, despite its advanced capabilities, is not a robust code model, as it can be attacked by adversarial codes generated by pre-trained code models, with a success rate of up to 49%. We believe our benchmark and findings provide valuable insights for developers and researchers in code intelligence, enabling more rigorous testing of the robustness of their code models.

ACKNOWLEDGMENTS

Jin WEN and Maxime Cordy are funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant BRIDGES/2022/IS/17437536. Yuejun Guo is funded by the European Union's Horizon Research and Innovation Programme, as part of the project LAZARUS (Grant Agreement n° 101070303). The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4, Article 81 (July 2018), 37 pages. https://doi.org/10.1145/3212695

[2] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source code authorship attribution using long short-term memory based networks. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 65–82.

[3] Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. 2020. Square attack: a query-efficient black-box adversarial attack via random search. In *European Conference on Computer Vision* (Virtual/Online). Springer Science+Business Media, Germany, 484–501.

[4] AtCoder Inc. 2017. Problem statement: B - shift only. https://atcoder.jp/contests/abc081/tasks/abc081_b. Accessed: 2024-12-30.

[5] AtCoder Inc. 2018. Problem statement: C - *3 or /2. https://atcoder.jp/contests/abc100/tasks/abc100_c. Accessed: 2024-12-30.

[6] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: learning to fix coding errors with a text-to-text transformer. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. https://proceedings.mlr.press/v139/berabi21a.html

[7] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. 2018. Practical black-box attacks on deep neural networks using efficient query mechanisms. In *Proceedings of the European Conference on Computer Vision* (Munich, Germany). Springer Science+Business Media, Germany.

[8] Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 84, 12 pages.

[9] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2018. Decision-based adversarial attacks: reliable attacks against black-box machine learning models. In *International Conference on Learning Representations* (Vancouver, Canada). OpenReview.net.

[10] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 39–57. https://doi.org/10.1109/SP.2017.49

[11] Boyuan Chen, Mingzhi Wen, Yong Shi, Dayi Lin, Gopi Krishnan Rajbahadur, and Zhen Ming Jiang. 2022. Towards training reproducible deep learning models. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2202–2214. https://doi.org/10.1145/3510003.3510163

[12] Francesco Croce and Matthias Hein. 2020. Minimally distorted adversarial examples with a fast adaptive boundary attack. In *ICML (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 2196–2205. https://proceedings.mlr.press/v119/croce20a.html

[13] Zeming Dong, Qiang Hu, Xiaofei Xie, Maxime Cordy, Mike Papadakis, and Jianjun Zhao. 2024. Importance guided data augmentation for neural-based code understanding. *arXiv preprint arXiv:2402.15769* (2024).

[14] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: white-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 2: Short Papers*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, 31–36. https://doi.org/10.18653/v1/P18-2006

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: a pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[16] GitHub, Inc. 2021. GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot Accessed: 2024-12-30.

[17] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR*, Yoshua Bengio and Yann LeCun (Eds.). San Diego, CA, USA.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 933–944. https://doi.org/10.1145/3180155.3180167

[19] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: pre-training code representations with data flow. In *ICLR*,. OpenReview.net.

[20] Qiang Hu, Yuejun Guo, Maxime Cordy, Xiaofei Xie, Lei Ma, Mike Papadakis, and Yves Le Traon. 2022. An empirical study on data distribution-aware test selection for deep learning enhancement. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4, Article 78 (July 2022), 30 pages. https://doi.org/10.1145/3511598

[21] Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Mike Papadakis, Lei Ma, and Yves Le Traon. 2023. CodeS: towards code model generalization under distribution shift. In *IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (Melbourne, Australia). IEE, 1–6. https://doi.org/10.1109/ICSE-NIER58687.2023.00007

[22] Yujin Huang, Terry Yue Zhuo, Qiongkai Xu, Han Hu, Xingliang Yuan, and Chunyang Chen. 2023. Training-free lexical backdoor attacks on language models. In *Proceedings of the ACM Web Conference 2023 (WWW)* (Austin, TX, USA) *(WWW '23)*, Ying Ding, Jie Tang, Juan F. Sequeda, Lora Aroyo, Carlos Castillo, and Geert-Jan Houben (Eds.). Association for Computing Machinery, New York, NY, USA, 2198–2208. https://doi.org/10.1145/3543507.3583348

[23] huggingface. 2016. Hugging Face – The AI community building the future. https://huggingface.co Accessed: 2024-12-30.

[24] Akshita Jha and Chandan K. Reddy. 2023. CodeAttack: code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*, Brian Williams, Yiling Chen, and Jennifer Neville (Eds.). AAAI Press, Article 1670, 9 pages. https://doi.org/10.1609/AAAI.V37I12.26739

[25] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2020. Is bert really robust? a strong baseline for natural language attack on text classification and entailment. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 05 (April 2020), 8018–8025. https://doi.org/10.1609/aaai.v34i05.6311

[26] Pang Wei Koh and Shiori Sagawa *et al.*,. 2021. WILDS: a benchmark of in-the-wild distribution shifts. In *International Conference on Machine Learning (ICML)*.

[27] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. *ICLR Workshop* (2017).

[28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJzIBfZAb

[29] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep learning meets software engineering: a survey on pre-trained models of source code. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, Lud De Raedt (Ed.). International Joint Conferences on Artificial Intelligence Organization, 5546–5555. https://doi.org/10.24963/ijcai.2022/775 Survey Track.

[30] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: multi-task learning with code-text transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, Royi Lachmy, Ziyu Yao, Greg Durrett, Milos Gligoric, Junyi Jessy Li, Ray Mooney, Graham Neubig, Yu Su, Huan Sun, and Reut Tsarfaty (Eds.). Association for Computational Linguistics, Online, 40–47. https://doi.org/10.18653/v1/2021.nlp4prog-1.5

[31] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss.

2021. CodeNet: a large-scale AI for code dataset for learning a diversity of coding tasks. (2021). https://openreview.net/forum?id=6vZVBkCDrHT

[32] Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. 2019. Generating natural language adversarial examples through probability weighted word saliency. In *ACL*. Association for Computational Linguistics, Florence, Italy, 1085–1097. https://doi.org/10.18653/v1/P19-1103

[33] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering* 25, 6 (2020), 5193–5254. https://doi.org/10.1007/S10664-020-09881-0

[34] Tran Van Sang, Tran Phuong Thao, Rie Shigetomi Yamaguchi, and Toshiyuki Nakata. 2022. Enhancing boundary attack in adversarial image using square random constraint. In *Proceedings of the 2022 ACM on International Workshop on Security and Privacy Analytics* (Baltimore, MD, USA) *(IWSPA '22)*. Association for Computing Machinery, New York, NY, USA, 13–23. https://doi.org/10.1145/3510548.3519373

[35] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating adversarial computer programs using optimized obfuscations. In *International Conference on Learning Representations*. https://openreview.net/forum?id=PH5PH9ZO_4

[36] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *30th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480. https://doi.org/10.1109/ICSME.2014.77

[37] Florian Tramèr, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. 2020. On adaptive attacks to adversarial example defenses. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/11f38f8ecd71867b42433548d1078e38-Abstract.html

[38] Faqeer ur Rehman and Madhusudan Srinivasan. 2023. Metamorphic testing for machine learning: applicability, challenges, and research opportunities. In *2023 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 34–39. https://doi.org/10.1109/AITEST58265.2023.00014

[39] Vivian van der Werf, Alaaeddin Swidan, Felienne Hermans, Marcus Specht, and Efthimia Aivaloglou. 2024. Teachers' Beliefs and Practices on the Naming of Variables in Introductory Python Programming Courses. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training, SEET@ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 368–379. https://doi.org/10.1145/3639474.3640069

[40] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[41] Zhilong Wang, Lan Zhang, Chen Cao, Nanqing Luo, Xinzhi Luo, and Peng Liu. 2024. How Does Naming Affect Language Models on Code Analysis Tasks? *Journal of Software Engineering and Applications* 17, 11 (2024), 803–816.

[42] Jin Wen. 2023. Black-box variable renaming attack for code model: benchmark and enhancement. https://sites.google.com/view/variable-attack-for-code-model Accessed: 2024-12-30.

[43] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

[44] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1482–1493. https://doi.org/10.1145/3510003.3510146

[45] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM Programming Languages* 4, OOPSLA, Article 162 (November 2020), 30 pages. https://doi.org/10.1145/3428230

[46] Jin Yong Yoo and Yanjun Qi. 2021. Towards improving adversarial training of nlp models. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. Association for Computational Linguistics, Punta Cana, Dominican Republic, 945–956. https://doi.org/10.18653/v1/2021.findings-emnlp.81

[47] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 1169–1176. https://doi.org/10.1609/aaai.v34i01.5469

[48] Jie Zhang, Wei Ma, Qiang Hu, Shangqing Liu, Xiaofei Xie, Yves Le Traon, and Yang Liu. 2023. A black-box attack on code models via representation nearest neighbor search. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 9706–9716. https://doi.org/10.18653/V1/2023.FINDINGS-EMNLP.649

[49] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine learning testing: survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 2 (2022), 1–36. https://doi.org/10.1109/TSE.2019.2962027

[50] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 915, 11 pages.