Evolutionary Multi-objective Optimization for Contextual Adversarial Example Generation

SHASHA ZHOU, University of Electronic Science and Technology of China, China and University of Exeter, United Kingdom
MINGYU HUANG, University of Electronic Science and Technology of China, China
YANAN SUN, Sichuan University, China
KE LI, University of Exeter, United Kingdom

The emergence of the ‘code naturalness’ concept, which suggests that software code shares statistical properties with natural language, paves the way for deep neural networks (DNNs) in software engineering (SE). However, DNNs can be vulnerable to certain human imperceptible variations in the input, known as adversarial examples (AEs), which could lead to adverse model performance. Numerous attack strategies have been proposed to generate AEs in the context of computer vision and natural language processing, but the same is less true for source code of programming languages in SE. One of the challenges is derived from various constraints including syntactic, semantics and minimal modification ratio. These constraints, however, are subjective and can be conflicting with the purpose of fooling DNNs. This paper develops a multi-objective adversarial attack method (dubbed `MOAA`), a tailored NSGA-II, a powerful evolutionary multi-objective (EMO) algorithm, integrated with `CodeT5` to generate high-quality AEs based on contextual information of the original code snippet. Experiments on 5 source code tasks with 10 datasets of 6 different programming languages show that our approach can generate a diverse set of high-quality AEs with promising transferability. In addition, using our AEs, for the first time, we provide insights into the internal behavior of pre-trained models.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Security and privacy → Software security engineering**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: multi-objective optimization, adversarial example, neural networks

ACM Reference Format:
Shasha Zhou, Mingyu Huang, Yanan Sun, and Ke Li. 2024. Evolutionary Multi-objective Optimization for Contextual Adversarial Example Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 101 (July 2024), 24 pages. https://doi.org/10.1145/3660808

1 INTRODUCTION

The large and growing body of successfully, widely used open source software systems such as Linux, MySQL, and Django, etc., has enabled billions of available code tokens and millions of instances of corresponding meta-data concerning authorship, bug-fixes and review processes. The availability of such ‘big code’ suggests a new, data-driven approach for software engineering tasks by uncovering statistical and distributional patterns in codes. In particular, since the past decade and beyond, there has been a growing interest in the idea of ‘code naturalness’, which could be

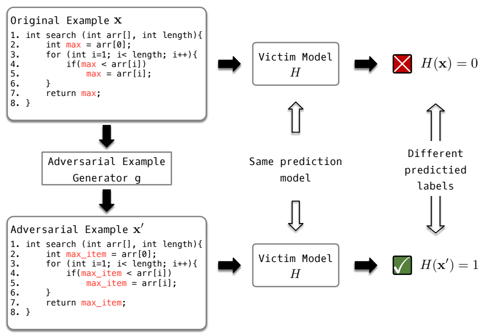
Authors’ Contact Information: Shasha Zhou, University of Electronic Science and Technology of China, Chengdu, China and University of Exeter, Exeter, United Kingdom, sz484@exeter.ac.uk; Mingyu Huang, University of Electronic Science and Technology of China, Chengdu, China, m.huang.gla@outlook.com; Yanan Sun, Sichuan University, Chengdu, China, ysun@scu.edu.cn; Ke Li, University of Exeter, Exeter, United Kingdom, k.li@exeter.ac.uk.

ACM 2994-970X/2024/7-ART101
https://doi.org/10.1145/3660808

traced back to the 'literate programming' concept of Knuth [34] in 1984. In [2], Allamanis *et al.* proposed the code naturalness hypothesis that link software corpora with natural language corpora.

On one hand, the statistics of large corpora of human communications have been extensively studied in the natural language processing (NLP) community, and advanced, highly curated models based on deep neural networks (DNNs) have been successfully applied to numerous applications including speech recognition [54, 65, 67], machine translation [10, 12, 35], and text classification [41, 85, 87, 88], to name a few. On the other hand, given the code naturalness hypothesis, one might then expect there are also underlying rich patterns in large code corpora, similar to natural language, which could be potentially exploited by DNNs. The first empirical evidence of this is reported in [28]. Hindle et al. showed that DNN models originally developed for NLP tasks were surprisingly effective for source code. To date, DNN-based models have achieved state-of-the-art (SOTA) performance in a variety of code-related tasks, e.g., vulnerability detection [3, 8, 26, 70], test case generation [5, 39], clone detection [7, 44, 71], code summarization [23, 92] and code translation [59, 63].

However, it has been widely recognized that DNNs are vulnerable to adversarial examples (AEs) [1, 72]. Generally speaking, they are crafted by adding small perturbations to benign inputs of the model, may easily fool DNNs, or at least lead to contradicting predictions [22, 43, 64]. This greatly impedes the usability of DNN models, since ideally the model should generate indistinguishable results for similar inputs. As a result, when DNNs are adopted, improving their robustness has become indispensable. There have been many works on AE generation, especially for image classification [74, 75, 80] and some NLP tasks [18, 50, 91]. For example, Goodfellow *et al.* [22] showed that inducing minor modifications to pixel values in the input image could steer the classification models into erroneous predictions. For NLP tasks, AEs can be generated by replacing certain tokens or characters in the text while maintaining the overall semantics [21, 32, 33, 88].


Original Example $\mathbf{x}$
Victim Model H
$H(\mathbf{x}) = 0$
Adversarial Example Generator g
Same prediction model
Different predictied labels
Adversarial Example $\mathbf{x}'$
Victim Model H
$H(\mathbf{x}') = 1$

```
int search (int arr[], int length){
    int max = arr[0];
    for (int i=1; i< length; i++){
        if(max < arr[i])
            max = arr[i];
    }
    return max;
}
```

```
int search (int arr[], int length){
    int max_item = arr[0];
    for (int i=1; i< length; i++){
        if(max_item < arr[i])
            max_item = arr[i];
    }
    return max_item;
}
```

Fig. 1. An example of AE generation on code task.

DNN robustness is also crucial to applications in software engineering, since code is often semantically brittle: even subtle changes (e.g., swapping function arguments) can drastically change the meaning of the code. In this case, there is an emerging demand for effective and efficient ways to generate high-quality AEs (Fig. 1 gives an illustrative example). However, this is considerably more challenging than both computer vision and NLP tasks. One of the reasons is that it must satisfy **syntactic constraints**, which stipulate that the generated adversarial code snippet must be compilable and executable. This is usually achieved by setting limitations such that perturbations could only be applied to certain positions like identifiers [53, 79, 83]. Another source of difficulty lies in that, the adversarial code snippet should meet **semantic constraints** [27, 29, 86], i.e., the perturbed code snippet should preserve the semantic meaning of the original one. For example, Yang et al. [78] proposed simultaneously considering the semantic constraint of generated example as well as preserving the syntactic similarity of original inputs. In addition, successful AEs should also possess **imperceptibility** [62]. In other words, it does not constitute a reasonable AE if the perturbation strength is too large, even though this always leads to a significant impact to the victim model.

As summarized in Table 1, existing works in the software engineering mainly formulate the AE generation as a single-objective optimization problem constrained by the various criteria. However, we argue that such formulation is not sufficiently effective because of the following three aspects.

Table 1. Related works for AE generation in software engineering community.

Methods	Syntactic	Semantic	Imperceptibility	Core Technique
Yefet *at al.* [79]	✓			Replaced variables in code using gradient information
Zhang *et al.* [83]	✓			Used the Metropolis-Hastings algorithm to find an optimal replacement for identifiers
Bielik & Vechev [6]	✓			Proposed a robust training strategy to abstain AE from deciding when uncertain
Srikant `et al.` [62]	✓		✓	Applied PGD [42] to generate adversarial examples of code
Rabin *et al.* [53]	✓			Applied six semantic preserving program transformations to produce new programs
Pour *et al.* [52]	✓	✓		Proposed a search-based testing framework for source code embedding
Nguyen *et al.* [48]	✓			Demonstrated deliberately forged data can spoil SOTA API recommender systems
Schuster *et al.* [61]	✓	✓		Demonstrated that neural code autocompleters are vulnerable to poisoning attacks
Henkel *et al.* [27]	✓	✓		Toke a gradient ascent step in the direction that maximizes model loss
Zhou *et al.* [89]	✓	✓		Substituted identifiers with K candidates using cosine distance for code comments generation
Yang *et al.* [78]	✓	✓		Leveraged natutalness-aware substitution to generate replacements for identifiers
Li *et al.* [38]	✓			Leveraged coding style patterns to enhance robust of source code authorship attribution models
Zhang *et al.* [82]	✓	✓		Applied rule-based constraints embedded within the attacking techniques
Zeng *et al.* [81]	✓	✓		Utilized word importance rank (WIR) to determine the substitution sequence of identifiers
Tian *et al.* [66]	✓	✓		Used code differences between target and reference inputs to guide AE generation
Du *et al.* [16]	✓	✓		Prioritized different types of statements and utilized beam search to generate AEs

- These constraints are too subjective to articulate a hard decision boundary. For instance, there is no clear-cut for two code snippets to be 'semantically' similar or not, and the tolerance for the perturbation strength is also ambiguous.
- These constraints can be conflicting with the purpose of fooling DNNs, which is usually quantified via an adversarial loss function as introduced in Section 2. In this case, it is difficult, if not impossible, to find a single optimal solution for all that ignores the trade-off nature.
- It is necessary to identify a collection of AEs that allow for meaningful statistical inference, in order to gain insights on the blind-spots of DNNs and identify problematic decision boundaries. However, the single-objective optimization problem setting can only offer one optimal solution at a time without diversity.

Bearing these considerations in mind, this paper proposes a multi-objective optimization problem (MOP) formulation for AE generation with sufficient diversity. In particular, we consider simultaneously optimizing the *adversarial loss* along with two types of constraints, namely *semantic similarity* and *perturbation strength*. To solve this problem, we develop multi-objective adversarial attack method (dubbed `MOAA`), a tailored NSGA-II [13], a seminal evolutionary multi-objective optimization algorithm, integrated with `CodeT5` [73] to generate high-quality AEs based on contextual information of the original code snippet. To adapt to the source code related tasks, we modify the definition of the classic dominance relation used in NSGA-II. Rather than obtaining a single optimal solution for a constraint optimization problem, `MOAA`, instead, lead to a set of solutions known as the Pareto front or Pareto optimial solutions. This could offer decision makers a diverse set of AEs with different trade-offs between the conflicting objectives and thus enable them to tailor solutions that resonate with different priorities and preferences.

To demonstrate the effectiveness of `MOAA`, we conduct comprehensive experiments on 5 source code tasks with 10 datasets of 6 different programming languages. Empirical results show that our approach can generate a diverse set of high-quality AEs with promising transferability, which is crucial for adversarial attacks. In addition, these generated AEs can help improve model robustness via adversarial fine-tuning. Last but not the least, using our AEs, for the first time, we provide insights into the internal behavior of pre-trained models.

In a nutshell, the main contributions of this paper are summarized as follows:

- We propose a novel perspective of formulating the AE generation problem as a MOP. This formulation allows for the creation of a range of AEs that demonstrate various trade-offs, enhancing the diversity and robustness of the generated AEs.
- We develop a novel multi-objective adversarial attack method, `MOAA`. It incorporates two key innovations: ► the integration of the NSGA-II algorithm to preserve diversity within

the population, helping to escape local optima; ► the employment of `CodeT5` to generate context-aware alternative candidates for identifiers, utilizing the contextual information from the original code snippet.

- We offer fresh insights into the internal mechanisms of pre-trained models through the analysis of the AEs generated by `MOAA`. These insights provide a deeper understanding of the vulnerabilities and behaviors of pre-trained models, contributing to the ongoing discourse in the field and paving the way for future research.

The rest of this paper is organized as follows. Section 2 provides a formal definition of the MOP considered in `MOAA`. Section 3 delineates the implementation of our proposed `MOAA`. The experimental setup is introduced in Section 4 and the results are discussed in Section 5. Finally, Section 6 and Section 7 discuss the related works and threats to validity respectively, while Section 8 concludes this paper and threads some light on future directions.

2 MULTI-OBJECTIVE ADVERSARIAL ATTACK

This section starts from a formal definition of the AE generation task considered in this paper, including its main goal and three relevant constraints. Then, we introduce how we formulate this task as a MOP as implemented in `MOAA`.

2.1 AE Generation Task

We define $H : \mathcal{X} \longrightarrow \mathcal{Y}$ as a predictive model, particularly a DNN classifier or generator in our context, where $\mathcal{X}$ and $\mathcal{Y}$ is the input and the output space, respectively. Given a code snippet $\mathbf{x} \in \mathcal{X}$, we focus on adversarial perturbations that slightly perturb $\mathbf{x}$ into $\mathbf{x}'$, such that $G(H, \mathbf{x}') = \textit{True}$. Here, G is a Boolean function that indicates the attack is successful or not, i.e., $\mathbf{x}'$ leads to an adverse prediction of the victim model. In addition, $\{C_1, \ldots, C_n\}$ are defined as a set of Boolean functions indicating whether the perturbation satisfies certain constraints. Generally speaking, an AE generation task considers the problem of searching for a perturbation from $\mathbf{x}$ to $\mathbf{x}'$ which fools H by both achieving some goal, as represented by $G(H, \mathbf{x}')$, and fulfilling each constraint $C_i(\mathbf{x}, \mathbf{x}')$. Formally, the task of AE generation can be defined as follows:

$$\text{Find } \mathbf{x}' \in \mathcal{X} \text{ such that:} \quad \begin{cases} G(H, \mathbf{x}') = \text{True} \\ \forall i \in \{1, \ldots, n\} : C_i(\mathbf{x}', \mathbf{x}) = \text{True}. \end{cases} \tag{1}$$

Goal Function. In practice, the goal G is often represented by a certain level of performance degradation. This could be quantified via an adversarial loss function, $\mathcal{L} : \mathcal{X} \times \mathcal{H} \longrightarrow \mathbb{R}$, where the exact definition would vary with tasks. In this, we consider two categories of tasks:

- **Code classification** (e.g., defect prediction [3, 8, 26, 70] and clone detection [7, 44, 71]). The output of H is a logit vector $\phi_H(\mathbf{x}) \in \mathbb{R}^K$ such that $Y = \text{argmax}_k\, \phi_H(\mathbf{x})_k$, where $k \in \{1, \cdots, K\}$ and $K > 1$ is the total number of classes. The adversarial loss can be defined as the predicted confidence of the true label Y_{true} given $\mathbf{x}'$:

$$\mathcal{L}(\mathbf{x}', H) = \phi_H(\mathbf{x}')_{Y_{\text{true}}}. \tag{2}$$

- **Code generation.** (e.g., code summarization [92] and code translation [59]). The output of H is a sequence of tokens $Y = \{y_i\}_{i=1}^m$. The adversarial loss $\mathcal{L}$ in this case is calculated as the similarity between the prediction $H(\mathbf{x}')$ and the reference output Y_{true} in terms of BLEU [51] or CodeBLEU [57] metric (see Appendix A.1 for more details):

$$\mathcal{L}(\mathbf{x}', H) = \text{(Code)BLEU}\left((\mathbf{x}'), Y_{\text{true}}\right). \tag{3}$$

Constraint Functions. In addition to defining the goal of the attack, we need to define some constraints that must be met to ensure the quality of the generated AEs, as introudced in Section 1.

- **Syntactic constraint.** AEs should be executable and compilable, which is similar to the grammatical constraint for AEs in NLP tasks [46]. However, while linguistic errors can be identified by grammar checkers [47], there does not exist analogous method in the context of programming languages. Existing literature usually employ pre-defined rules to make perturbations (e.g., identifier renaming and permute statement) to avoid syntactic errors [27, 53]. Following this rigor, this paper only considers perturbations on the identifier names, and we thereby do not have explicit expressions for this type of constraint.
- **Semantic constraint.** AEs should preserve the semantic meaning of the original code snippet. It is formulated as the following constraint:

$$C_1(\mathbf{x}', \mathbf{x}) = S(\mathbf{x}', \mathbf{x}) > \sigma_1, \tag{4}$$

  where $S(\mathbf{x}', \mathbf{x})$ evaluates the semantic similarity between $\mathbf{x}'$ and $\mathbf{x}$ in an embedded space while σ_1 is a pre-defined threshold.
- **Imperceptibility constraint.** The number of perturbed tokens between the original code snippet and AEs needs to be as small as possible. This constraint is evaluated as the proportion of the modified tokens undertaken by the $\mathbf{x}'$ relative to $\mathbf{x}$:

$$C_2(\mathbf{x}', \mathbf{x}) = D(\mathbf{x}', \mathbf{x}) > \sigma_2. \tag{5}$$

  The function $D(\mathbf{x}', \mathbf{x})$ is used to assess the level of imperceptibility between $\mathbf{x}'$ and $\mathbf{x}$, taking into account the granularity of either tokens of characters. We then compare this assessment to a predefined threshold σ_2.

2.2 AE Generation as Multi-objective Optimization Problem

Given a code snippet $\mathbf{x} \in \mathcal{X}$, an AE, denoted as $\mathbf{x}' \in \mathcal{X}$, is generated by replacing identifier names of $\mathbf{x}$ without compromising the syntactic constraint. In this paper, we formulate the AE generation task as the following three-objective optimization problem:

$$\min_{\mathbf{x}' \in \mathcal{X}} \big(f_1(\mathbf{x}'), f_2(\mathbf{x}'), f_3(\mathbf{x}')\big). \tag{6}$$

- The first objective function is the adversarial loss (AL) as defined in equations (2) and (3):

$$f_1(\mathbf{x}') = \mathcal{L}(\mathbf{x}', H). \tag{7}$$

- The second objective function is the semantic similarity (SS). It is defined as the cosine similarity, denoted as $\cos(\cdot,\cdot)$, between $\mathbf{x}'$ and $\mathbf{x}$ in the embeded space:

$$f_2(\mathbf{x}') = \sum_{i=1}^{N_{\text{var}}} 1 - \cos(v_i', v_i), \tag{8}$$

  where v_i is an identifier name in $\mathbf{x}$, $i \in \{1, \ldots, N_{\text{var}}\}$ and N_{var} is the number of identifier names in $\mathbf{x}$. Note that we use $1 - \cos(\cdot)$ to transform this objective function as a minimization problem. A smaller value of SS indicates a higher level of similarity between $\mathbf{x}'$ and $\mathbf{x}$.
- The last objective function is the modification rate (MR) that evaluates the ratio of the perturbed tokens (i.e., the number of tokens in $\mathbf{x}'$ different $\mathbf{x}$) w.r.t. the total number of tokens:

$$f_3(\mathbf{x}') = \frac{|\mathbf{x}' - \mathbf{x}|}{|\mathbf{x}|} \times 100\%, \tag{9}$$

  where $|*|$ indicates the number of tokens in the set. The lower the f_3 is, the less perturbations caused by the adversarial attack.

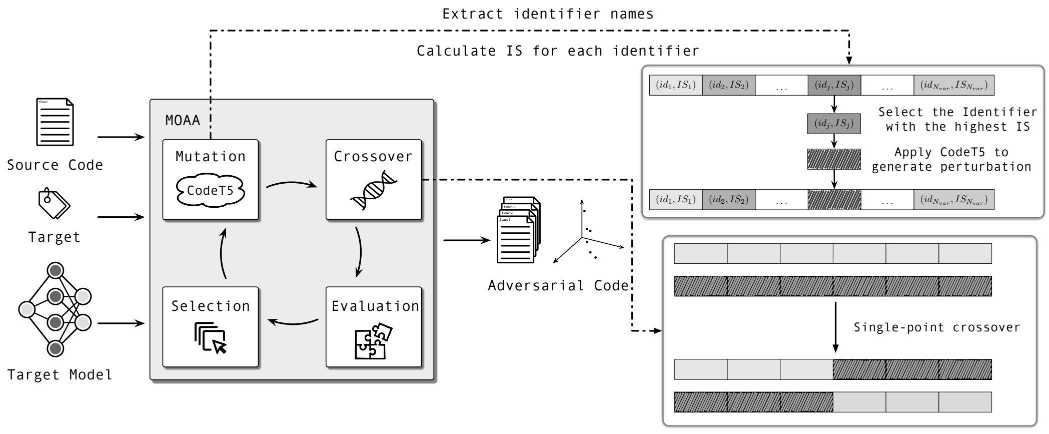


Fig. 2. The high-level overview of the conceptual architecture of MOAA.

3 METHODOLOGY

Fig. 2 gives a high-level overview of the conceptual architecture of the proposed MOAA. It consists of four main components: 1) mutation; 2) crossover; 3) evaluation, and 4) selection. During the mutation and crossover steps, we develop tailored reproduction operators to generate new code examples. Following this, we compute objective values for each generated example in evaluation step. Finally, the selection step determines the composition of the next population. This section begins by providing a systematic exposition of the implementation of MOAA. Then, we delineate the working mechanism of the crux of this paper in the mutation step, i.e., evaluating the significance of each identifiers in the original example through importance scores and predicting replaced identifier names using a pre-trained model.

3.1 Implementation of Our Proposed MOAA

The pseudo code of our proposed MOAA is given in Algorithm 1, which is based on NSGA-II [13]. During the initialization stage, MOAA starts with evaluating the significance of each identifier within the input code snippet $\mathbf{x}$ by importance scores, denoted as **IS** (lines 2-3). This will be described in Section 3.2. In lines 4-7, we apply the mutation operator described in Section 3.3 upon each of the N identifier names selected with the probability proportional to the **IS**. Thereafter, we come up with a population of initialized solutions, denoted as $\mathcal{P}^1 = \{\hat{\mathbf{x}}\}_{i=1}^N$.

In the main loop, a single-point crossover operator is applied to generate another N offspring (lines 9-12). This process involves selecting two parent solutions $\hat{\mathbf{x}}^1$ and $\hat{\mathbf{x}}^2$ from the parent population $\mathcal{P}^t$. A random crossover point within the set of identifiers (denoted as **ids**) is chosen, and the identifiers before and after this point are swapped between the two parents. The newly generated solutions, $\hat{\mathbf{x}}^1_{new}$ and $\hat{\mathbf{x}}^2_{new}$, are added to the offspring population Q. Then, we apply the mutation operator to each solution in Q (lines 14-17). To select the elite population $\mathcal{P}^{t+1}$ for the next generation, we apply the non-dominated sorting method to select the best N solutions from the combined population of the generated offspring Q and $\mathcal{P}^t$ (lines 18-20). Different from the seminal NSGA-II, we redefine the Pareto dominance relation to adapt to the context of AE generation.

Definition 1. A solution $\mathbf{x}^1$ is said to *dominate* the other solution $\mathbf{x}^2$, denoted as $\mathbf{x}^1 \preceq \mathbf{x}^2$ when:

(1) If $\mathbf{x}^1$ or $\mathbf{x}^2$ are not both AEs, and $f_1(\mathbf{x}^1) < f_1(\mathbf{x}^2)$.
(2) Or if $\mathbf{x}^1$ and $\mathbf{x}^2$ are both AEs, $f_i(\mathbf{x}^1) \leq f_i(\mathbf{x}^2)$, $\forall i \in \{1,2,3\}$ and $\exists j \in \{1,2,3\}$ such that $f_j(\mathbf{x}^1) < f_j(\mathbf{x}^2)$.

This redifined dominance relation is pivotal in our context, which concenteates on the Pareto-optimal region containing the successful AEs. In scenarios where two solutions are not both successful AEs, preference is given to the solution with the lower AL, indicating closer to the desired Pareto-optimal region. When both solutions are successful AEs, we give equal weight to the three objectives. This adjusted balance among the objectives is iteratively exploited by the non-dominated sorting method in the selection operation of NSGA-II algorithms.

Algorithm 1: Pseudo code of MOAA

```
Input: original code $\mathbf{x}$, original label $Y_{true}$, target model $H$, population size $N$, prediction model $T$, the
       number of recommand identifiers $n$
Output: population $\mathcal{P}^t$
1  $t \leftarrow 1, \mathcal{P}^t \leftarrow \emptyset$;
2  Extract identifiers ids from original code snippet $\mathbf{x}$;
3  Apply Algorithm 2 on $\mathbf{x}$ to obtain the important scores IS;
4  for $i \leftarrow 1, \cdots, N$ do
5      Select an identifier name $id$ with the probability proportional to the IS;
6      Apply the mutation operation upon $id$ to obtain a perturbed code snippet $\hat{\mathbf{x}}^i$;
7      $\mathcal{P}^t \leftarrow \mathcal{P}^t \bigcup \{\hat{\mathbf{x}}^i\}$;
8  while stopping criterion is not met do
9      $Q \leftarrow \emptyset$;
10     for $i \leftarrow 1, \cdots, \frac{N}{2}$ do
11         Randomly pick up $\hat{\mathbf{x}}^1$ and $\hat{\mathbf{x}}^2$ from $\mathcal{P}^t$;
12         Apply the crossover operation upon $\hat{\mathbf{x}}^1$ and $\hat{\mathbf{x}}^2$ to obtain an offspring $\hat{\mathbf{x}}^1_{new}$ and $\hat{\mathbf{x}}^2_{new}$;
13         $Q \leftarrow Q \bigcup \{\hat{\mathbf{x}}^1_{new}, \hat{\mathbf{x}}^2_{new}\}$;
14     for $i \leftarrow 1, \cdots, N$ do
15         Select an identifier name $id$ from $\hat{\mathbf{x}}^i_{new}$ in $Q$ with the probability proportional to the IS;
16         Apply the mutation operation upon $id$ to obtain a perturbed code snippet $\tilde{\mathbf{x}}^i_{new}$;
17         $Q \leftarrow (Q \setminus \{\hat{\mathbf{x}}^i_{new}\}) \bigcup \{\tilde{\mathbf{x}}^i_{new}\}$;
18     $Q \leftarrow Q \bigcup \mathcal{P}^t$;
19     Apply the non-dominated sorting operation to obtain the next iteration population $\mathcal{P}^{t+1}$;
20     $t \leftarrow t + 1$;
21 return $\mathcal{P}^t$;
```

3.2 Importance Score

As introduced in Section 3.1, in order to improve the efficiency of MOAA, we evaluate the significance of each identifier in the original example when conducting the mutation. Inspired by importance-based adversarial attacks in NLP [21, 30, 33, 56, 88], which consider that each token in the text carries varying importance towards the classification outcome of deep neural networks, we propose the use of an importance score (IS) to assess the significance of each identifier in the code. The definition of the IS for the i^{th} identifier in the original code snippet $\mathbf{x}$ depends on the type of tasks. For classification tasks, it is defined as follows:

$$IS_i = \phi_H(\mathbf{x})_{Y_{\text{true}}} - \phi_H(\mathbf{x}_i^*)_{Y_{\text{true}}}, \tag{10}$$

where $\mathbf{x}_i^*$ is new code snippet by masking the i^{th} identifier in $\mathbf{x}$.

As for generation tasks, IS is calculated by the changed value of BLEU [51] or CodeBLEU [57].

$$IS_i = \text{(Code)BLEU}(H(\mathbf{x}_i^*), Y_{\text{true}}) - \text{(Code)BLEU}(H(\mathbf{x}), Y_{\text{true}}). \tag{11}$$

Algorithm 2 demonstrates how MOAA computes the *IS* for each identifier in the original code snippet. Firstly, a parser[1] is employed to extract the list of identifiers, denoted as **ids**, from the code snippet **x** (line 2). Each obtained identifier in **ids** is then masked individually to obtain its corresponding importance score (lines 3-6). Specifically, all positions of the identifier are replaced by '<UNK>' in **x**, resulting in a modified code snippet $\mathbf{x}_i^*$ (line 4). Next, the modified $\mathbf{x}_i^*$ is fed into the target model H to obtain its confidence in predicting the original label. The difference in confidence between **x** and $\mathbf{x}_i^*$ w.r.t. the original label serves as the importance score for the i^{th} identifier (line 5). We assume that a higher value of *IS* implies a greater influence of the identifier on the prediction outcome, thus granting priority to identifiers with a significant perturbation effect.

Algorithm 2: Importance Score Calculation

```
Input: original code snippet $\mathbf{x}$, original label $Y$, target model $H$
Output: importance score IS, identifier list ids
1 IS $\leftarrow \emptyset$, $i \leftarrow 0$;
2 Extract identifiers ids from original code snippet $\mathbf{x}$;
3 for id in ids do
4     replace id in $\mathbf{x}$ by '<UNK>' to generate new code snippet $\mathbf{x}_i^*$;
5     Calculate $IS_i$ by equation (10) or equation (11);
6     $i \leftarrow i + 1$;
7 return IS, ids
```

3.3 Model-Based Identifier Name Prediction

After determining the identifier name to replace, the next step is to choose the specific word that will serve as the replacement. In this paper, we leverage the function of masked language prediction of pre-trained models to predict a set of possible replacements. Specifically, we employ CodeT5 [73] for this purpose for the reason that it has been pre-trained on the masked identifier prediction (MIP) task and is able to predict a suitable identifier names within the context of code snippet. After a fine-tuning in the MIP downstream task, CodeT5 merely provides a predictive role throughout the adversarial attack process, generating the perturbed code.

The process of generating the perturbed code can be divided into four steps. First, all positions of the specified identifier in **x** are masked by '<extra_id_0>', which is a unique mask token in the T5Tokenizer. Subsequently, this code snippet is tokenized by the T5Tokenizer into a sequence of sub-token index. CodeT5 then returns the top-k identifier names for this masked token. Finally, MOAA randomly selects one of the identifier name to replace the original identifier.

It is worthy to note that the main difference of MOAA regarding the existing works lies in the input representation for identifier names. Unlike ALERT [78], which splits identifier names into sub-tokens and feeds them to the model, our method takes the entire code segment as input. By doing so, the model can make predictions based on the contextual information within the code snippet. Consequently, our approach enables the generation of more realistic combinations of identifier names compared to existing rule-based methods, which are limited by the information contained within the existing identifier names.

[1] https://tree-sitter.github.io/tree-sitter/

4 EXPERIMENTAL SETUP

To comprehensively evaluate MOAA and illustrate its advantages over other established AE generation methods, we conduct large-scale empirical experiments on 3 victim DNN models for 5 source code tasks, using 5 evaluation metrics that focus on complementary aspects.

4.1 Source Code Tasks and Datasets

To keep comparisons thorough, fair and reproducible, we consider a broad collection of publicly-available source-code tasks that are widely concerned in the SE community [77]. Here we provide a brief sketch of them, and more detailed information can be found in Appendix A.2.

- **`Defect detection`.** It is a classification task that aims to ascertain whether a given source code contains defects that may be used to attack software systems.
- **`Clone detection`.** This classification task aims to identify similarity in operational semantics between different code snippets.
- **`Authorship attribution`.** It is another classification task that invovles analyzing coding patterns, stylistic features, and other idiosyncratic attributes in the source code to accurately predict the author's identity.
- **`Code translation`.** This is a generation task that focuses on the automated conversion of source code from one programming platform to another while ensuring semantic equivalence and functional integrity.
- **`Code summarization`.** Its objective is to generate accurate and concise summaries of code functionalities or algorithms that can provide quick insights into the code's purpose.

4.2 Victim Models

In this paper, we choose three representative pre-trained model in the code intelligence community, as the victim models to validate the efficacy of AE generation methods.

- **CodeBERT [19].** It is a bimodal pre-trained model that builts on the BERT architecture, specifically tailored for the domain of programming languages and NLP. It is designed to understand and generate code by learning from a vast corpus of programming language documentation and source code, and thus bridges the gap between human and programming languages.
- **GraphCodeBERT [24].** GraphCodeBERT extends the capabilities of CodeBERT by integrating the inherent syntactic and semantic structure of code into its learning process. It is able to capture more fine-grained code semantics and dependencies, and thus offers enhanced performance in more sophisticated source code tasks.
- **CodeT5 [73].** It is a unified text-to-text model designed specifically for programming tasks. It considers some unique characteristics of programming language, such as token types and code semantics conveyed by developer-assigned identifiers.

To ensure fair results, we need to make sure that all three victim models are adequately optmized to their expected performance. In doing so, we followed the suggestions in [78] (for CodeBERT and GraphCodeBERT) and [73] (for CodeT5) to fine-tune all three models before our experiments.

4.3 Evaluation Metrics

In order to systematically evaluate the performance of different AE generation methods, we consider the following five metrics in this paper, each of which focuses on different performance aspects. For explicit mathematical definitions, please refer to Appendix A.3.

- **Attack success rate (ASR).** It measures the effectiveness of adversarial attacks by calculating the proportional of successful AEs in the total trials. Specifically:

- For classification task, an AE is successful if it leads to inconsistencies in model prediction (i.e., the predicted label alters).
- For generation task, following the practice in the NLP community [17, 60], we say an AE is successful if their BLEU or CodeBLEU scores decrease by over 50%.

- **Average adversarial loss (AAL).** It represents the average value of the adversarial loss. Lower values indicate greater confidence in the model's incorrect predictions.
- **Average semantic similarity (ASS).** It captures the average semantic similarity between original and adversarial examples. Lower values indicate less semantical changes.
- **Average modification rate (AMR).** It indicates the average proportion of altered tokens in successful AEs. Lower values imply more modifications.
- **Average query count (AQC).** It reflects the expected number of queries to the victim models needed to generate a successful AE. For MOAA, as it generates a *collection* of AEs with diverse trade-offs, here we evalute the AQC based on the number of queries it takes to generate the first AE to allow for a fair comparison. Lower values signify higher attack effeiciency.

5 RESULTS AND DISCUSSIONS

We seek to answer the following four research questions (RQs) through our experimental evaluation:

- **RQ1:** What is the relationship between the three proposed objectives?
- **RQ2:** How is the performance of MOAA compared to other established methods?
- **RQ3:** What are the effectiveness of AEs generated by MOAA?
- **RQ4:** Can we interpret how model works from the perspective of adversarial attack?

The **RQ1** is designed to illustrate the necessity of multi-objectivizing AE generation. Specifically, we explore whether this problem could be simplified by concentrating on a singular objective (i.e., the objectives exhibit positive correlations). We investigate **RQ2** to compare the performence of MOAA against various state-of-the-art (SOTA) algorithms in diverse scenarios. Through **RQ3**, we aim to assess the quality of the AEs generated by MOAA. Lastly, **RQ4** is designed to provide some insights on the blind-spots of DNNs and identify problematic decision boundaries.

5.1 Relationship Between Objective Functions

5.1.1 Methods. In order to answer RQ1, we design a greedy search method to collect a set of AL, SS and MR when perturbing identifiers in code snippet, and use scatter plot of matrices (SPLOM) to visualize the correlation of them. More specifically, we apply a parser to extract a set of identifier names from each test dataset. For each example in the test data, we perform the following steps:

- **Identifier selection.** Randomly choose 30 identifiers from the extracted set of identifiers.
- **Perturbation and evaluation.** Generate 30 perturbed examples by replacing the original identifier name with the highest **IS** value, utilizing the selected identifiers. Subsequently, the corresponding objective function values of the perturbed examples are evaluated. The example with smallest AL value is retained, and concurrently record the corresponding objective values.
- **Iterative perturbation.** Iterates for the next identifier in the descending order of **IS** values.

This iterative process is repeated for all identifiers within the given example. After collecting these objective values, we employ SPLOM to plot scatter plots for each pair objectives and frequency histogram. Additionally, Pearson correlation metric is used to calculate the correlation of each pair objectives. This approach allows us to see the interrelationships between three objectives.

5.1.2 Results and Analysis. Fig. 3 gives a visual interpretation of the pairwise relationships among three objectives while employing a greedy search to minimize AL on CodeBERT for the defect detection task. A more comprehensive analysis of these relationships in the context of different models and tasks are available in Appendix B.1. Overall, it is clear to see that the increase of AL leads to a decline of SS and MR, while the increase of SS leads to the elevation of MR. Notably, for the same task, the distribution of objectives and their histograms exhibit relative similarity, while it shows significant variations across difference tasks. This can be attributed to the fact that all victim models capture unique task-specific characteristics of tasks. These observations imply the conflict relationship between our designed objectives, i.e., an increase in AL can lead to a decrease of SS and MR. This reinforces the significance of multi-objectivizing AE generation.

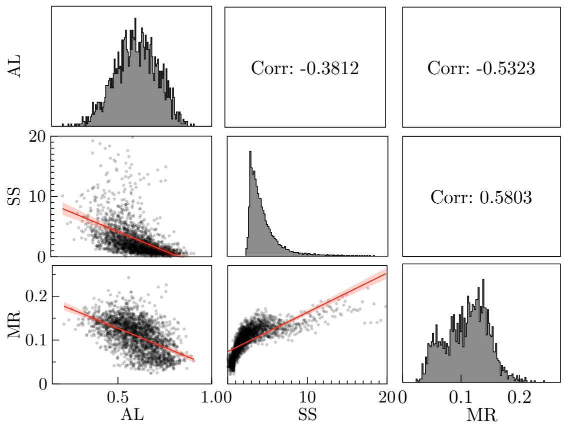


Fig. 3. Scatter plot of matrices (SPLOM) visualizes the correlation of AL, SS and MR values for CodeBERT on defect detection. Scatter plots below the diagonal show the linear regression that fit for the two objectives, shown on the diagonal frequency histograms, while Pearson correlation values (Corr) are shown above the diagonal.

Response to *RQ1*: By conducting an exploratory analysis the interrelationships among the three objectives, we find that AL conflicts with the other two objectives. This observation reinforces the significance of employing an evolutionary multi-objective algorithm to generate AEs.

5.2 Performance Comparison with the SOTA Baseline Algorithms

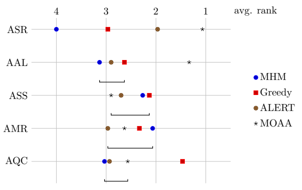


Fig. 4. Critical difference (CD) diagram of the performance of MOAA and baseline algorithms under the 5 metrics. Each point indicates the average rank (lower is better) of performance across all tasks and victim models. Points grouped by the horizontal lines are not significantly different.

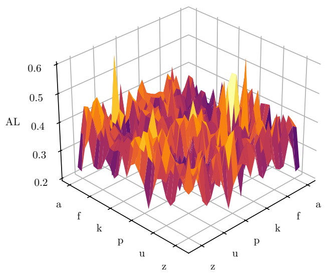


Fig. 5. A projected landscape of the performance of AEs in objective space for CodeBERT on defect detection.

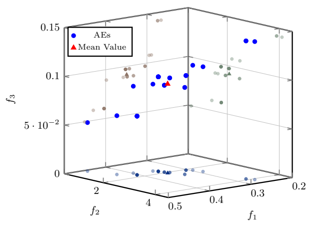


Fig. 6. Visualization of the distribution of generated AEs by MOAA for defect detection dataset in the objective space.

5.2.1 Methods. To demonstrate the superiority of MOAA, we compare its performance with three well-estalished AE generation algorithms, namely MHM [83], Greedy Attack [78] and ALERT [78], all of which use a single-objective problem formuation (see more details in Appendix A.4). We assess the performance of MOAA and these algorithms using the setup described in Section 4. In particular, for each dataset and victim model, to evaluate algorithm performance on arbitrary code snippets, we randomly pick up 1, 000 snippets to serve as the original inputs. The only exception is the authorship attribution task, which only contains 132 examples, and we use all of them in the experiments. To ensure the robustness and reproducibility of our results, we conduct a total of 25 repeated experiments for each scenario using 5 distinct random seeds, where for each seed we repeat the experiment for 5 runs. Note that for all baseline algorithms, the search space is configured following [78], which considers the semantic similarity when replacing identifier names. We use the same parameter settings for population size $N_{\text{pop}} = 30$ and maximum number of iterations $n_{\text{iter}} = 5 \times N_{\text{var}}$, where N_{var} is the number of extracted identifier names, for all the algorithms.

To avoid a cluttered table, we present the results of the comparison using a **critical different (CD)** diagram, which can depict the statistical significance of the performance differences between algorithms as measured by the 5 metrics introduced in Section 4.3 across all tasks in a compact manner. For reference, we also provide the raw results in Appendix B.2. In addition to comparing individual metrics, we are also interested in whether MOAA is able to optimize all three objectives simultaneously. In doing so, we calculate the **dominance rate** of MOAA over the baseline algorithms, which quantifies the proportion of evaluated code snippets for which MOAA is able to generate AEs that achieve superior performance across all three objectives compared to AEs generated by the baseline algorithms.

5.2.2 Results and Analysis. As shown in the CD diagram in Fig. 4, MOAA significantly outperforms all 3 baseline algorithms in terms of attack efficacy (ASR and AAL scores) while consumes similar number of queries as MHM and ALERT (as indicated by the AQC score). We postulate that the reason could be the rugged *landscape* surface of source code tasks (see, e.g., Fig. 5), in which local optima are prevalent. Consequently, algorithms that solely focus on the AL objective are more likely to be trapped by such local optima. In contrast, MOAA maintains population diversity through Pareto dominance relation and crowding distance, which enhances the likelihood of escaping the local optima. Another potential reason for this is that MOAA employs CodeT5 to generate candidate identifiers, which significantly expands the search space (see Appendix B.3 for more details).

In addition to attack effeiciency, MOAA is also competitive in maintaining the semantic similarity and modification rate of the generated AEs. Note that in Fig. 4, MOAA does not exhibit statistically significant differences in terms of ASS and AMR scores. This is because MOAA generates a set of AEs with diverse trade-offs between the three objectives. When calculating these metrics, we aggregate the values across the entire population, and thus the reported ASS and AMR represent the centric values of the whole population, which can be obscured by extreme AEs that solely focusing on optimizing the adversarial loss (see Fig. 6 for an example and Appendix B.2 for detailed discussions).

To better illustrate this, we compare each individual AE generated by MOAA with baseline methods. When considering all three objectives simultaneously, we find that in all scenarios, for a considerable fraction of code inputs, AEs generated by MOAA are superior in all aspects to (i.e., dominate) those produced by baseline methods (see Fig. 7). Notably, for the attack on CodeT5 in the authorship attribution task, in 90% cases MOAA can generate AEs that dominate the baseline algorithms.

Response to *RQ2*: *By evaluating on a diverse set of tasks and different victim models, we illustrate that* MOAA *is more efficient and natural in AE generation than any other existing approaches. Notably, while maintaining a diverse trade-offs between different objectives in the generated AEs,* MOAA *is able to dominate the baselines across all three objectives with considerable chance.*

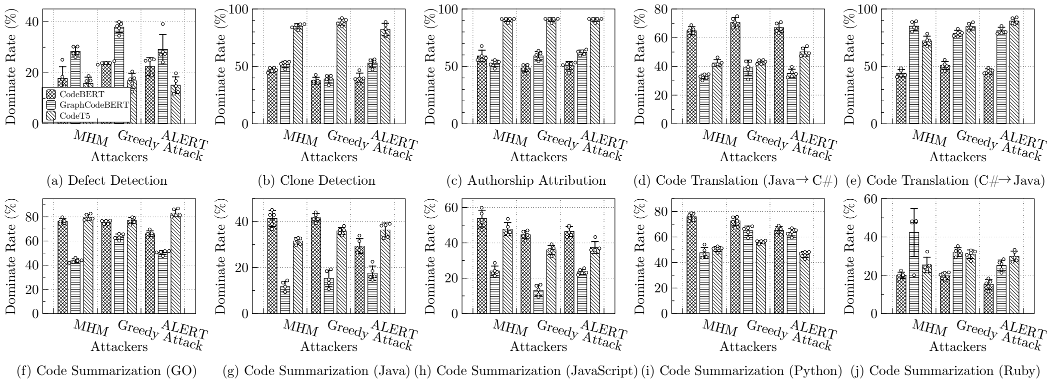


Fig. 7. Dominance rate of MOAA in AE generation across all three objectives over the baseline algorithms.

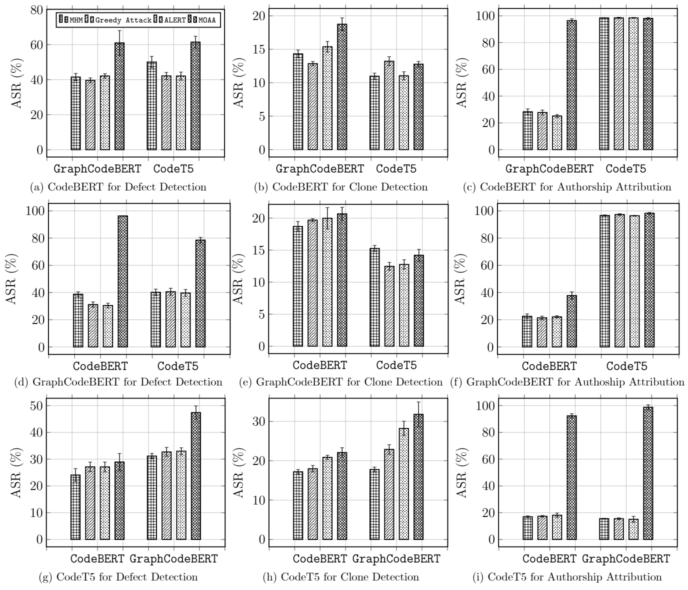


Fig. 8. Bar charts of the attack success rate of transferring AEs generate from one model to the other on the `defect detection`, `clone detection`, and `authorship attribution` tasks. Fig. (a) illustrates the success rates of adversarial attacks on GraphCodeBERT and CodeT5 models using AEs generated through MHM, Greedy Attack, ALERT, and MOAA algorithms on CodeBERT models.

5.3 Comparison of the Generated AEs

5.3.1 Methods. To address RQ3, this subsection aims to investigate the effectiveness of the generated AEs from the following three aspects.

Transferability. It evaluates the usefulness of an AE for attacking the models beyond the target model. To this end, we first randomly pick up 1, 000 examples from the test dataset of the `defect detecion` and `clone detection` tasks, and the entire test dataset for the `authorship attribution` task. Then, for a given target model, we apply MHM, Greedy Attack, ALERT and MOAA to generate AEs, respectively. Thereafter, the generated AEs are used as inputs to feed into the other two

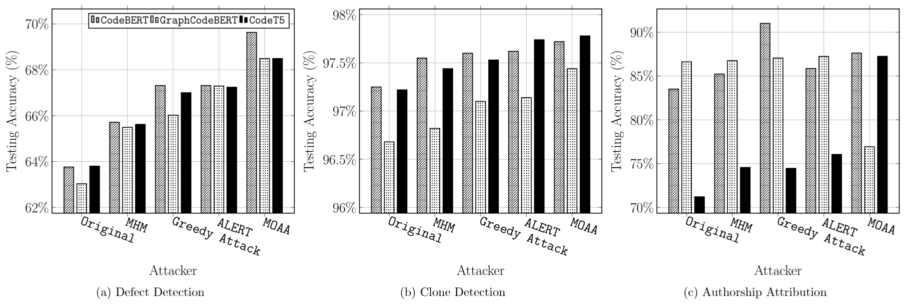


Fig. 9. Testing accuracy of CodeBERT, GraphCodeBERT, and CodeT5 after adversarial fine-tuning with AEs generated using MHM, Greedy Attack, ALERT, and MOAA methods on the defect detection, clone detection, and authorship attribution tasks.

models. To evaluate the transferability, we evaluate the success rates of the generated examples that successfully attack the target models. Each experiment is repeated 5 times while CodeBERT, GraphCodeBERT and CodeT5 are used as the base models for AE generation in a round-robin manner.

Adversarial Fine-tuning. It can reveal the defects of models and improve the robustness. To this end, we first randomly pick up 1,000 examples from the training dataset for each dataset mentioned in Section 4.1, excluding the authorship attribution task which employs the entire data dataset. Then, we apply MHM, Greedy Attack, ALERT and MOAA to generate AEs accordingly. Thereafter, the generated AEs are used to augment the training set for fine-tuning CodeBERT, GraphCodeBERT and CodeT5 models with parameters mentioned in Section 4.2. The testing accuracy of the fine-tuned models is used as the measure of the effectiveness of the fine-tuning.

Subjective Study. We also evaluate the naturalness of the AEs generated by each algorithm by conducting a user study. Following previous works [66, 78], we invite 10 non-author participants who possess a Bachelor/Master degree in Computer Secience. For this study, to accommodate the preferences of the participants, we select tasks of different programming languages for each of them. To alleviate recognition burden, we then filter the dataset to exclude code snippets longer than 200 tokens. To make comparison fair, we only select examples that can be successfully attacked by all four algorithms. We randomly pick up 100 examples from the remaining code snippets. For each of these examples, we take the AEs generated by each algorithm[2]. Therefore, we ask the participants to grade 400 $\langle \mathbf{x}, \mathbf{x}' \rangle$ pairs blindly in a 5-point rating scale (for the detailed grading criteria, see Appendix B.5). They are instructed to focus on whether the replaced identifiers are natural in the context of the code snippet.

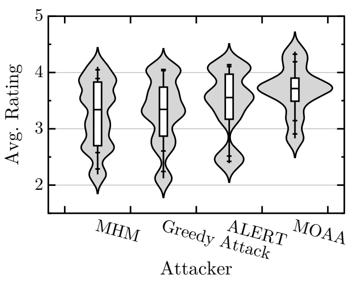


Fig. 10. Results of the subjective study to evaluate naturalness of AEs. The y-axis corresponds to the average ratings, where 5 means the highest naturalness and 1 means the lowest.

5.3.2 Results and Analysis. From the results shown in Fig. 8, we can clearly see that the generalization performance of the AEs generated by our proposed MOAA is better than that of the other baseline algorithms. A major reason for this could be the more diverse AE populations, which

[2]For MOAA we randomly select one AE from the generated population.

provide more chances for deceiving the victim models. The advantage of such diversity is further illustrated by the superior adversarial fine-tuning performance with MOAA (Fig. 9), possibly due to a better converage for model blind points. In addition, the results of the subjective study as shown in Fig. 10 show that MOAA achieves higer ratings in terms of AE naturalness, which is a direct benefit of considering semantic similarity as an optimization objective. Another possible factor for this advance is the incorporation of contextual information when replacing the identifier names. As an illustration, Table 2 presents an examplar AE population generated by MOAA, including the extremal AE on each objective. Additionally, it is intriguing to note that we find no particular preference of participants on AEs with different trade-offs between objectives. This underscores the importance of diversity in AE generation, since users may have different preferences for distinct usecases.

Table 2. Example of population of AEs generated by MOAA when attacking CodeBERT model for the `defect detection` dataset.

Exapmles	**Input**

Original	static void lumRangeToJpeg16_c(int16_t *_dst, int width)
	{
	int i;
	int32_t *dst = (int32_t *) _dst;
	for (i = 0; i < width; i++)
	dst[i] = (FFMIN(dst[i],30189«4)*19077 - (39057361«4))»14;
	}

AE_1 (with smallest AL)	_dst→fromLum;
	i→dstIndex;
	dst→dst_ptr

AE_2 (with smallest SS)	dst→dst_ptr
AE_3 (with smallest MR)	_dst→_lum
AE_4 (with smallest MR)	_dst→_src
AE_5	_dst→jpeg16; dst→range
...	...

Response to *RQ3*: *The transferability of the AEs generated by* MOAA *is surpasses that of the other baseline algorithm on the* `defect detection`, `clone detection`, *and* `authorship attribution` *tasks. Moreover, the AEs generated by* MOAA *have a better chance to improve the predictive performance of the victim model via adversarial fine-tuning. In addition, the AEs generated by* MOAA *are more natural than those generated by the other baseline algorithms.*

5.4 Explaining the Generated AEs

5.4.1 Methods. To address RQ4, we design the following three experiments that aim to provide some insights from the generated AEs.

Attention Analysis. One of the most attractive advantages of the attention mechanism [68] is its ability to identify the information in an input most pertinent to accomplishing a task. we attempt to scrutinize the self-attention weights, fundamental components in pre-training Transformer-based models to investigate how subtle perturbations, albeit challenging to detect, can lead to considerable performance degradation. Specifically, at each Transformer layer, we derive a set of attention weights denoted as α over the input code, where $\alpha_{i,j} > 0$ represents the attention from i-th code token to j-th token. This yields 144 attention maps for both CodeBERT and GraphCodeBERT, both of which are composed of 12 layers of Transformer encoder with 12 attention heads. Additionally, CodeT5, consisting of 12 layers of Transformer encoders and 12 layers of decoders with 12 attention heads, provides us with 288 attention maps. Subsequently, we employ cosine similarity as a metric to assess the similarity between the attention maps of original and AEs, which shows what happend when perturbations add to original examples. With this aim in mind, we randomly select 5, 000 pairs of original examples and AEs generated by `MOAA` on the `defect detection`, `clone detection`, and `authorship attribution` tasks. To ensure uniform attention maps' size, we truncate all the long code sequences within the length of 512. Following this, we employ heatmaps to visually represent the disparities present in each attention map.

Feature Importance. This experiment primary aims to answer, from the perspective of feature importance, the reason why certain examples can be easily generated AEs, while others cannot. We employ local interpretable model-agnostic explanations (LIME) [58], a widely recognized technique for elucidating individual predictions produced by black-box machine learning (ML) models. LIME accomplishes this by systematically introducing variations of the input data into the ML model and subsequently revealing the pivotal features influencing the predictions. Therefore, we proceed by randomly selecting 1, 000 original exampls that are enable to generating AEs using `MOAA`, along with another set of 1, 000 examples that can not generate AEs. Subsequently, we employ LIME to analyze the top 10 influential tokens affecting the prediction outcomes. This enables us to quantify the similarity between the perturbed identifier and the crucial features under examination.

Adversarial Fine-tuning. Given that `MOAA` can generate a set of AEs for each input, it becomes crucial to select an optimal solution for decision maker, such as the knee-point method [9, 49]. This experiment allows us to leverage these data to elucidate which specific AEs are more conducive to enhancing the robustness of DNNs. In this part, we first randomly pick 1, 000 examples from the training dataset on the `defect detection`, `clone detection`, and `authorship attribution` tasks to generate AEs by `MOAA`. Then, we divide each set of AEs into three distinct groups based on the values of AL, SS and MR. These categorized sets are subsequently combined with the original training dataset for the purpose of fine-tuning the `CodeBERT`, `GraphCodeBERT` and `CodeT5` models.

5.4.2 Results and Analysis.

Attention Analysis. Fig. 11 shows the dynamics of attention weights as they evolve in response to perturbation for the `CodeBERT`, `GraphCodeBERT`, and `CodeT5` models across various code-related tasks. In general, these models exhibit task-dependent variations, yet certain commonalities emerge. Notably, both the `CodeBERT` and `GraphCodeBERT` models are based on the BERT [15] architecture, and it is apparent that the attention patterns within the first four layers ($L1$ to $L4$) of these models display a degree of consistency. Inspired by Wan *et al.* [69], this observation can be attributed to the initial layers of these models primarily capturing positional and content information of source code tokens, with deeper attention aligning with the abstract syntax tree motif structure. This suggests that probably shallow models are potentially better robustness. In contrast, the `CodeT5` model, rooted in the T5 [55] architecture, exhibits distinct behavior from `CodeBERT` and `GraphCodeBERT` models. A comparative analysis of encoder and decoder layers reveals that perturbation exerts a greater difference to the encoder layers, whereas the decoder layers are relatively less affected.

Furthermore, examining the differences between the same models across various tasks reveals consistent patterns. For instance, head 10 ($H10$) in layer 2 ($L2$) of `CodeBERT` exhibits greater dissimilarity across all three tasks compared to other heads at the same layer, indicated by a redder hue. While in `GraphCodeBERT`, layer 1 ($L1$) shows a higher degree of similarity between head 10 ($H10$) and 12 ($H12$), denoted by a bluer hue. This pattern suggests that, following pre-training, each attention mechanism within the model captures fixed information, and fine-tuning exerts a relatively limited impact on attention patterns.

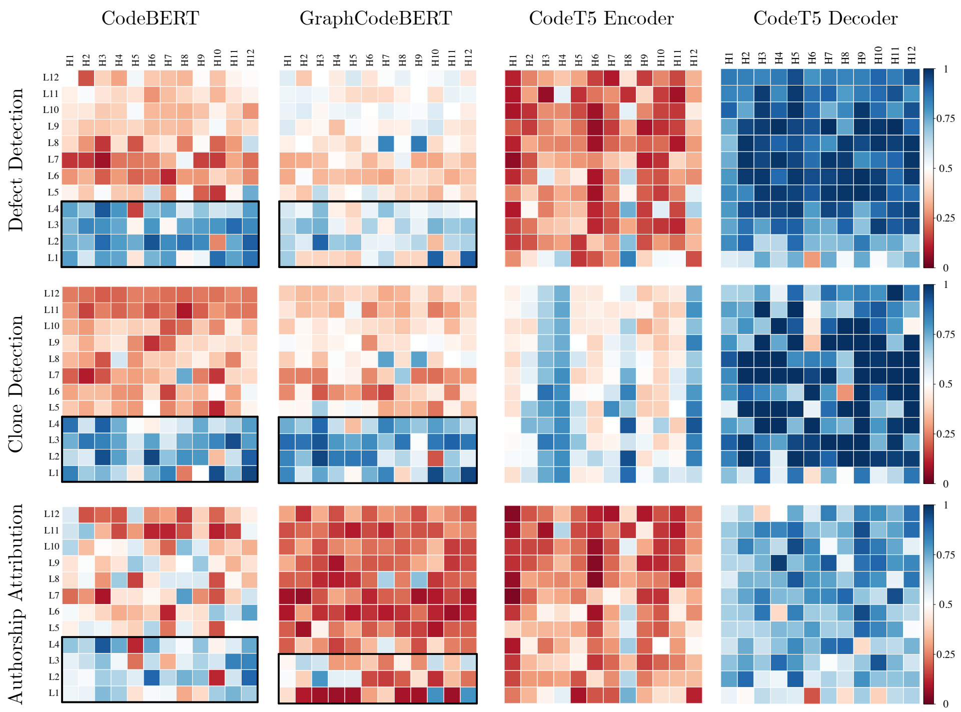
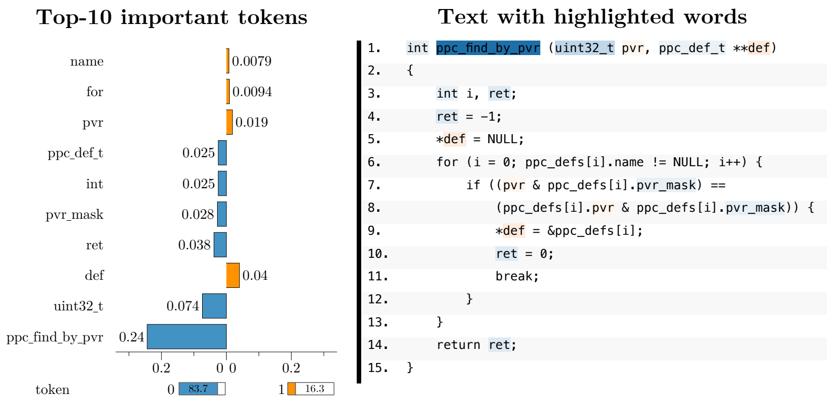


Fig. 11. The average cosine similarity scores representing the attention distribution between AE and original example have been computed for each layer-head in the CodeBERT, GraphCodeBERT and CodeT5 models across various tasks, including defect detection, clone detection and authorship attribution. Higher scores indicate a greater degree of similarity, while lower scores signify a greater degree of dissimilarity.



```
int ppc_find_by_pvr (uint32_t pvr, ppc_def_t **def)
{
    int i, ret;
    ret = -1;
    *def = NULL;
    for (i = 0; ppc_defs[i].name != NULL; i++) {
        if ((pvr & ppc_defs[i].pvr_mask) ==
            (ppc_defs[i].pvr & ppc_defs[i].pvr_mask)) {
            *def = &ppc_defs[i];
            ret = 0;
            break;
        }
    }
    return ret;
}
```

Fig. 12. The example of LIME explanation on the defect detection task.

Feature Importance. Fig. 12 presents an example of LIME applied to elucidate the significance of tokens in a code snippet on the defect detection task. The left panel shows that the 10 most influential tokens on the DNN's prediction. Notably, the identifier name "pvr" and "def" are observed to impact the prediction, yielding a result of "1", signifying the presence of a defect. Conversely, the identifier "ret" is found to influence the prediction outcome of "0", indicative of non-defect.

Fig. 13 provides a statistical analysis of the percentage of examples capable of successfully generating AEs through the perturbation of identifier names and those unable to do so. We use the LIME to assess the inclusion of identifiers within the top 10 tokens. In the subset of example where AEs could successfully be generated, LIME indicates that in most cases (84%), the top 10 tokens exerting the most significant influence on prediction contain identifiers. Conversely, in 16% of these successfully cases, identifiers are absent from the top 10 important tokens. Among the examples incapable of generating AEs, 38% include identifiers in LIME prediction, while the remaining 62% do not. This result implies that the ability to generate AEs by perturbing identifier names is linked to the perturbation of the tokens whose features that hold the greater influence in prediction. Furthermore, we find that LIME tends to assign heightened importance to identifiers when explaining feature importance. This may be attributed to the frequent recurrence of identifiers in the code. These findings can help improve the efficiency for AE generation through prefering more importance features.

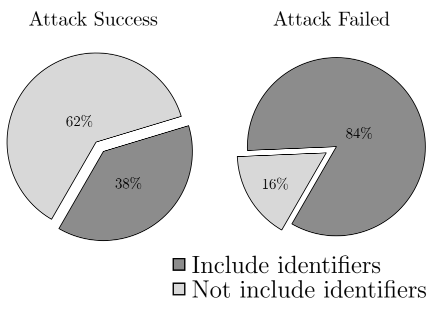


Fig. 13. Statistical analysis of identifiers inclusion in LIME of successful and failed AE generation.

Adversarial Fine-tuning. Fig. 14 compares the testing accuracy after adversarial fine-tuning of models utilizing generated AEs by MOAA. Notably, this adversarial fine-tuning process enhances the model's performance. For the AL factor, the most substantial improvement in testing accuracy is observed when AEs are selected with medium values. This could be rationalized by considering that excessively larger AL values result in minimal deviation of the decision boundary, while overly small AL values lead to excessive example deviation, causing a significant divergence in the decision boundary. As for the SS and MR, our findings indicate that the Low SS and Low MR groups, representing AEs closely resembling the original example, exhibit the most pronounced enhancements in performance. Consequently, it is plausible to conclude that in a population of AEs generated by MOAA, a decision maker exhibiting moderate AL values along with diminished SS and MR values may yield a more feasible approach for enhancing the model's robustness.

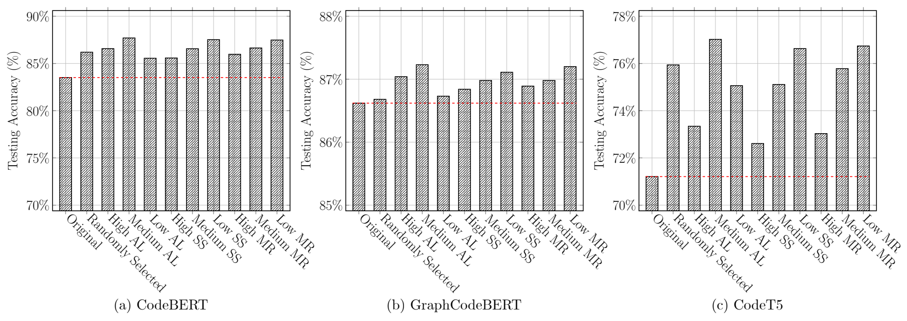


Fig. 14. Testing accuracy of CodeBERT, GraphCodeBERT and CodeT5 after adversarial fine-tuning with selected AEs generated using MOAA on `authorship attribution` task.

Response to *RQ4*: *In this subsection, we provide insights into AEs from three aspects, i.e. attention analysis, feature importance and adversarial training. Different models capture different attention patterns. The success of AE generation is relative to feature importance explained through LIME. Our case study on* `authorship attribution` *shows that a medium AL and low SS and MR are more likely to enhance the model's performance in adversarial fine-tuning.*

6 RELATED WORK

This section reviews the related work on evolutionary multi-objective optimization and AE generation on source code.

6.1 Evolutionary Multi-objective Optimization

Over the past several decades, many efforts have been dedicated to developing evolutionary multi-objective optimization algorithms (MOEAs). Generally, MOEAs can be categorized into three classes: (1) Pareto-dominance-based MOEAs [13, 14], (2) decomposition-based MOEAs [31, 36, 37, 84], and (3) indicator-based MOEAs [4, 90]. The first category uses Pareto dominance to rank solutions and maintain diversity. For example, NSGA-II [13] employs crowding distance to measure diversity during selection, following the ranking of solutions through the fast non-dominated sorting method. The second category decomposes the target MOP into a set of single-objective subproblems and the solutions of each subproblem forms the final solution set. For example, MOEA/D [84] and RVEA [11] both perform decomposition by means of a set of predefined weight or reference vectors and optimize the subproblems simultaneously. The third category, such as IBEA [90] and HypE [4], candidates solutions that contribute more to the performance indicator are selected.

In theory, our proposed MOAA framework is algorithm-agnostic, i.e., any MOEA can be applied to serve the purpose of adversarial attack. Given the simplicity and widely reported effectiveness of NSGA-II, it is chosen as the backbone of MOAA in this paper. As outlined in Section 3.1, the task of AE generation concentrates primarily on the Pareto-optimal region, which encompasses the successful AEs. This presents significant challenges in defining the weight vector for decomposition-based MOEAs, as well as establishing a performance indicator for indicator-based MOEAs.

6.2 Adversarial Example Generation for Deep Code Models

The generation of AEs for deep code models has garnered increasing attention in recent years, whith methodologies broadly categorized into white-box and black-box approaches. In the white-box setting, attackers have full access to the victim model, including its architecture and parameters. For example, Yefet *et al.* [79] introduce DAMP, which leverages gradient information to substitute identifier names in source code. Srikant *et al.* [62] formulate the AE generation as an optimization problem and apply projected gradient descent [42] for joint optimization. Besides, Zhang *et al.* [82] develop CARROT, a method with rule-based constraints to refine the search process. These methods, while insightful, fall short in real-world applicability and suffer from model-specific limitations.

Conversely, in the black-box scenarios, where attackers are restricted to model queries, various strategies have been explored. MHM [83] introduce an iterative identifier substitution based on Metropolis-Hastings (M-H) sampling [25, 45]. Rabin *et al.* [53] show that GGNNs [20] can learn general representations of semantically equivalent programs by variable renaming. Nguyen *et al.* [48] and Schuster *et al.* [61] have demonstrated the susceptibility of DNNs to semantic manipulations in applications like API recommendation and code autocompletion. Furthermore, Yang *et al.* [78] design an attack based on greedy algorithm and genetic algorithm, named ALERT, to simultaneously consider the semantic constraints of generated examples as well as preserving the syntactic similarity of original inputs. Zhou *et al.* [89] introduce ACCENT, a method that selects K candidates through cosine distance for identifier substitution, aiming to create adversarial code snippets specifically for the task of generating code comments. Moreover, Li *et al.* [38] propose RoPGen, which identifiers unique coding styles as a defense mechanism, whereas, Du *et al.* [16] emphasize the effectiveness of context specific identifier substitution, proposing an innovative method that combine statement prioritization and beam search to generate AEs.

As summarized in Table 1, existing works in the software engineering community mainly formulate the AE generation as a single-objective optimization problem, subject to specific constraints. This approach neglects the inherent trade-off between the objective and constraints [74, 76]. In contrast, our study pioneers the formulation of AE generation as a MOP, delicately balancing the trade-off between the AL, SS, and MR. Moreover, our work introduces novel perspectives for uncovering insights on the blind-spots of DNNs.

7 THREATS TO VALIDITY

The main threat to validity lies in the selection of victim models, datasets and parameter settings. For victim models, we select three open-sourced SOAT models, which are extensively studied in existing works on AE generation [16, 66, 78]. Therefore, it is not guaranteed that `MOAA` can generate high-quality AEs for other models. For datasets, we use the CodeXGLUE benchmark [40], including 10 datasets of 6 different programming languages. However, the datasets are not comprehensive enough to cover all the possible scenarios in real-world applications. Parameter settings such as the number of population size and the termination criterion condition can lead to different results. We follow the settings in the peer algorithms [78, 83] to achieve a fair comparison. However, the settings may not be optimal for `MOAA`. Besides, the randomness in the initialization of the population and the mutation operations can also lead to different results. We run each algorithm for 25 times and report the average results to mitigate the randomness. Besides, for RQ4, the calibration of both attention map and LIME might be threats to validation.

8 CONCLUSION

This paper proposes a multi-objective optimization problem formulation for AE generation, which optimizing adversarial loss, semantic similarity and perturbation strength simultaneously. Subsequently, we develop a new multi-objective adversarial attack, dubbed `MOAA` to generate high-quality AEs based on contextual information of the original code snippet. Through experimenting on 5 source code tasks with 10 datasets of 6 different programming languages, we show that `MOAA` can generate a diverse set of high-quality AEs with promising transferability. Furthermore, we show that these generated AEs could enhance the predictive performance through adversarial fine-tuning. Additionally, this work provides a pioneering exploration into the internal behavior of pre-trained models from three aspects, including attention analysis, feature importance and adversarial fine-tuning. Artifacts of this paper are available at: https://github.com/COLA-Laboratory/MOAA. Appendix is available at: https://zenodo.org/records/11071846.

ACKNOWLEDGMENT

This work was supported in part by the UKRI Future Leaders Fellowship under Grant MR/S017062/1 and MR/X011135/1; in part by NSFC under Grant 62376056 and 62076056; in part by the Kan Tong Po Fellowship (KTP\R1\231017); the Alan Turing Fellowship; and the EPSRC under Grant 2404317.

REFERENCES

[1] Naveed Akhtar and Ajmal S. Mian. 2018. Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey. *IEEE Access* 6 (2018), 14410–14430.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4 (2018), 81:1–81:37.

[3] Bander Alsulami, Edwin Dauber, Richard E. Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks. In *ESORICS'17: Proc. of the 22nd European Symposium on Research in Computer Security*, Vol. 10492. 65–82.

[4] Johannes Bader and Eckart Zitzler. 2011. HypE: An Algorithm for Fast Hypervolume-Based Many-Objective Optimization. *Evol. Comput.* 19, 1 (2011), 45–76.

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *CoRR* (2022).

[6] Pavol Bielik and Martin T. Vechev. 2020. Adversarial Robustness for Code. In *ICML'20: Proc. of the 37th International Conference on Machine Learning*, Vol. 119. 896–907.

[7] Lutz Büch and Artur Andrzejak. 2019. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *SANER'19: Proc. of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. 95–104.

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Software Eng.* 48, 9 (2022), 3280–3296.

[9] Renzhi Chen and Ke Li. 2021. Knee Point Identification Based on the Geometric Characteristic. In *SMC'21: Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 764–769.

[10] Xilun Chen, Ahmed Hassan Awadallah, Hany Hassan, Wei Wang, and Claire Cardie. 2019. Multi-Source Cross-Lingual Model Transfer: Learning What to Share. In *ACL'19: Proc. of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. 3098–3112.

[11] Ran Cheng, Yaochu Jin, Markus Olhofer, and Bernhard Sendhoff. 2016. A Reference Vector Guided Evolutionary Algorithm for Many-Objective Optimization. *IEEE Trans. Evol. Comput.* 20, 5 (2016), 773–791.

[12] Xin Cheng, Shen Gao, Lemao Liu, Dongyan Zhao, and Rui Yan. 2022. Neural Machine Translation with Contrastive Translation Memories. In *EMNLP'22: Proc. of the Conference on Empirical Methods in Natural Language Processing*. 3591–3601.

[13] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2000. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimisation: NSGA-II. In *PPSN'00: Proc. of the Parallel Problem Solving from Nature*, Vol. 1917. 849–858.

[14] Kalyanmoy Deb and Himanshu Jain. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Trans. Evol. Comput.* 18, 4 (2014), 577–601.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT'19: Proc. of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186.

[16] Xiaohu Du, Ming Wen, Zichao Wei, Shangwen Wang, and Hai Jin. 2023. An Extensive Study on Adversarial Attack against Pre-trained Models of Code. In *ESEC/FSE'23: Proc. of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 489–501.

[17] Javid Ebrahimi, Daniel Lowd, and Dejing Dou. 2018. On Adversarial Examples for Character-Level Neural Machine Translation. In *COLING'18: Proc. of the 27th International Conference on Computational Linguistics*. 653–663.

[18] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: White-Box Adversarial Examples for Text Classification. In *ACL'18: Proc. of the 56th Annual Meeting of the Association for Computational Linguistics*. 31–36.

[19] 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 *EMNLP'20: Findings of the Association for Computational Linguistics*. 1536–1547.

[20] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *ICLR'19: Proc. of the 7th International Conference on Learning Representations.*

[21] Siddhant Garg and Goutham Ramakrishnan. 2020. BAE: BERT-based Adversarial Examples for Text Classification. In *EMNLP'20: Proc. of the 2020 Conference on Empirical Methods in Natural Language Processing*. 6174–6181.

[22] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *ICLR'15: Proc. of the 3rd International Conference on Learning Representations.*

[23] Jian Gu, Pasquale Salza, and Harald C. Gall. 2022. Assemble Foundation Models for Automatic Code Summarization. In *SANER'22: Proc. of the IEEE International Conference on Software Analysis, Evolution and Reengineering*. 935–946.

[24] 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'21: Proc. of the 9th International Conference on Learning Representations.*

[25] W Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. (1970).

[26] Jingxuan He, Luca Beurer-Kellner, and Martin T. Vechev. 2022. On Distribution Shift in Learning-based Bug Detectors. In *ICML'22: Proc. of the International Conference on Machine Learning*, Vol. 162. 8559–8580.

[27] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas W. Reps. 2022. Semantic Robustness of Models of Source Code. In *SANER'22: Proc. of the IEEE International Conference on Software Analysis, Evolution and Reengineering*. 526–537.

[28] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *ICSE'12: Proc. of the 34th International Conference on Software Engineering*. 837–847.

[29] Hossein Hosseini and Radha Poovendran. 2018. Semantic Adversarial Examples. In *CVPR'18: Proc. of the 2018 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 1614–1619.

[30] Yu-Lun Hsieh, Minhao Cheng, Da-Cheng Juan, Wei Wei, Wen-Lian Hsu, and Cho-Jui Hsieh. 2019. On the Robustness of Self-Attentive Models. In *ACL'19: Proc. of Conference of the Association for Computational Linguistics*. 1520–1529.

[31] Mingyu Huang and Ke Li. 2024. A Survey of Decomposition-Based Evolutionary Multi-Objective Optimization: Part II – A Data Science Perspective. *CoRR* (2024).

[32] Robin Jia and Percy Liang. 2017. Adversarial Examples for Evaluating Reading Comprehension Systems. In *EMNLP'17: Proc. of the 2017 Conference on Empirical Methods in Natural Language Processing*. 2021–2031.

[33] 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. In *AAAI'20: Proc. of the 34th AAAI Conference on Artificial Intelligence*. 8018–8025.

[34] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.

[35] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *ACL'20: Proc. of the Association for Computational Linguistics*. 7871–7880.

[36] Ke Li. 2024. A Survey of Decomposition-Based Evolutionary Multi-Objective Optimization: Part I-Past and Future. *CoRR* (2024).

[37] Ke Li, Kalyanmoy Deb, Qingfu Zhang, and Sam Kwong. 2015. An Evolutionary Many-Objective Optimization Algorithm Based on Dominance and Decomposition. *IEEE Trans. Evol. Comput.* 19, 5 (2015), 694–716.

[38] Zhen Li, Qian (Guenevere) Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In *ICSE'22: Proc. of the 44th IEEE/ACM 44th International Conference on Software Engineering*. 1906–1918.

[39] Muyang Liu, Ke Li, and Tao Chen. 2020. DeepSQLi: deep semantic learning for testing SQL injection. In *ISSTA'20: Proc. of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 286–297.

[40] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *NeurIPS'21: Proc. of the Neural Information Processing Systems Track on Datasets and Benchmarks*.

[41] Changze Lv, Jianhan Xu, and Xiaoqing Zheng. 2023. Spiking Convolutional Neural Networks for Text Classification. In *ICLR'23: Proc. of the Eleventh International Conference on Learning Representations*.

[42] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *ICLR'18: Proc. of the 6th International Conference on Learning Representations*.

[43] Rishabh Maheshwary, Saket Maheshwary, and Vikram Pudi. 2021. Generating Natural Language Attacks in a Hard Label Black Box Setting. In *AAAI'21: Proc. of the 35th AAAI Conference on Artificial Intelligence*. 13525–13533.

[44] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2022. Modeling Functional Similarity in Source Code With Graph-Based Siamese Networks. *IEEE Trans. Software Eng.* 48, 10 (2022), 3771–3789.

[45] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edwards Teller. 1953. Equation of State Calculations by Fast Computing Machines. (1953), 1087–1092.

[46] John X. Morris, Eli Lifland, Jack Lanchantin, Yangfeng Ji, and Yanjun Qi. 2020. Reevaluating Adversarial Examples in Natural Language. In *EMNLP'20: Findings of the Association for Computational Linguistics*. 3829–3839.

[47] Daniel Naber et al. 2003. A rule-based style and grammar checker. (2003).

[48] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. 2021. Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee*f*. In *ASE'21: Proc. of the 36th IEEE/ACM International Conference on Automated Software Engineering*. 253–265.

[49] Haifeng Nie, Huiru Gao, and Ke Li. 2020. Knee Point Identification Based on Voronoi Diagram. In *SMC'20: Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 1081–1086.

[50] Nicolas Papernot, Patrick D. McDaniel, Ananthram Swami, and Richard E. Harang. 2016. Crafting adversarial input sequences for recurrent neural networks. In *MILCOM'16: Proc. of the 2016 IEEE Military Communications Conference*. 49–54.

[51] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *ACL'02: Proc. of the 40th Annual Meeting of the Association for Computational Linguistics*. 311–318.

[52] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. 2021. A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding. In *ICST'21: Proc. of the 14th IEEE Conference on Software Testing, Verification and Validation*. 36–46.

[53] Md. Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Inf. Softw. Technol.* 135 (2021), 106552.

[54] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2022. Robust Speech Recognition via Large-Scale Weak Supervision. *CoRR* (2022).

[55] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.

[56] Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. 2019. Generating Natural Language Adversarial Examples through Probability Weighted Word Saliency. In *ACL'19: Proc. of the 57th Conference of the Association for Computational Linguistics*. 1085–1097.

[57] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* (2020).

[58] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *KDD'16: Proc. of the 22nd Conference on Knowledge Discovery and Data Mining*. 1135–1144.

[59] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *ICLR'22: Proc. of the 10th International Conference on Learning Representations*.

[60] Sahar Sadrizadeh, Ljiljana Dolamic, and Pascal Frossard. 2023. TransFool: An Adversarial Attack against Neural Machine Translation Models. *CoRR* (2023).

[61] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *USENIX'21: Proc. of the 30th USENIX Security Symposium*. 1559–1575.

[62] 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 *ICLR'21: Proc. of the 9th International Conference on Learning Representations*.

[63] Marc Szafraniec, Baptiste Rozière, Hugh Leather, Patrick Labatut, François Charton, and Gabriel Synnaeve. 2023. Code Translation with Compiler Representations. In *ICLR'23: Proc. of the 11th International Conference on Learning Representations*.

[64] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *ICLR'14: Proc. of the 2nd International Conference on Learning Representations*.

[65] Yun Tang, Hongyu Gong, Ning Dong, Changhan Wang, Wei-Ning Hsu, Jiatao Gu, Alexei Baevski, Xian Li, Abdelrahman Mohamed, Michael Auli, and Juan Miguel Pino. 2022. Unified Speech-Text Pre-training for Speech Translation and Recognition. In *ACL'22: Proc. of the 60th Annual Meeting of the Association for Computational Linguistics*. 1488–1499.

[66] Zhao Tian, Junjie Chen, and Zhi Jin. 2023. Adversarial Attacks on Neural Models of Code via Code Difference Reduction. *CoRR* (2023).

[67] Hsiang-Sheng Tsai, Heng-Jui Chang, Wen-Chin Huang, Zili Huang, Kushal Lakhotia, Shu-Wen Yang, Shuyan Dong, Andy T. Liu, Cheng-I Lai, Jiatong Shi, Xuankai Chang, Phil Hall, Hsuan-Jui Chen, Shang-Wen Li, Shinji Watanabe, Abdelrahman Mohamed, and Hung-yi Lee. 2022. SUPERB-SG: Enhanced Speech processing Universal PERformance Benchmark for Semantic and Generative Capabilities. In *ACL'22: Proc. of the 60th Annual Meeting of the Association for Computational Linguistics*. 8479–8492.

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS'17: Advances in Neural Information Processing Systems*. 5998–6008.

[69] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What Do They Capture? - A Structural Analysis of Pre-Trained Language Models for Source Code. In *ICSE'22: Proc. of the 44th IEEE/ACM 44th International Conference on Software Engineering*. 2377–2388.

[70] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering*. 297–308.

[71] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *SANER'20: Proc. of the 27th Conference on Software Analysis, Evolution and Reengineering*. 261–271.

[72] Xuezhi Wang, Haohan Wang, and Diyi Yang. 2022. Measure and Improve Robustness in NLP Models: A Survey. In *NAACL'22: Proc. of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4569–4586.

[73] 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 *EMNLP'21: Proc. of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[74] Phoenix Williams and Ke Li. 2023. CamoPatch: An Evolutionary Strategy for Generating Camoflauged Adversarial Patches. In *NeurIPS'23: Proc. in Neural Information Processing Systems*.

[75] Phoenix Neale Williams and Ke Li. 2023. Black-Box Sparse Adversarial Attack via Multi-Objective Optimisation CVPR Proceedings. In *CVPR'23: Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12291–12301.

[76] Phoenix Neale Williams, Ke Li, and Geyong Min. 2023. Sparse Adversarial Attack via Bi-objective Optimization. In *EMO'23: Proc. of the Evolutionary Multi-Criterion Optimization - 12th International Conference*, Vol. 13970. Springer, 118–133.

[77] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv.* 54, 10s (2022).

[78] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-trained Models of Code. In *ICSE'22: Proc. of the 44th IEEE/ACM 44th International Conference on Software Engineering*. 1482–1493.

[79] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 162:1–162:30.

[80] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2019. Adversarial Examples: Attacks and Defenses for Deep Learning. *IEEE Trans. Neural Networks Learn. Syst.* 30, 9 (2019), 2805–2824.

[81] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *ISSTA'22: Proc. of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 39–51.

[82] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards Robustness of Deep Program Processing Models - Detection, Estimation, and Enhancement. *ACM Trans. Softw. Eng. Methodol.* 31, 3 (2022), 50:1–50:40.

[83] 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 *AAAI'20: Proc. of the Conference on Artificial Intelligence*. 1169–1176.

[84] Qingfu Zhang and Hui Li. 2007. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Trans. Evol. Comput.* 11, 6 (2007), 712–731.

[85] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level Convolutional Networks for Text Classification. In *NIPS'15: Advances in Neural Information Processing Systems*. 649–657.

[86] Zhengli Zhao, Dheeru Dua, and Sameer Singh. 2018. Generating Natural Adversarial Examples. In *ICLR'18: Proc. of the 6th International Conference on Learning Representations*.

[87] Shasha Zhou, Ke Li, and Geyong Min. 2022. Adversarial example generation via genetic algorithm: a preliminary result. In *GECCO'22: Proc. of the Genetic and Evolutionary Computation Conference*.

[88] Shasha Zhou, Ke Li, and Geyong Min. 2022. Attention-Based Genetic Algorithm for Adversarial Attack in Natural Language Processing. In *PPSN'22: Proc. of the Parallel Problem Solving from Nature*, Vol. 13398. Springer, 341–355.

[89] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2022. Adversarial Robustness of Deep Code Comment Generation. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 60:1–60:30.

[90] Eckart Zitzler and Simon Künzli. 2004. Indicator-Based Selection in Multiobjective Search. In *PPSN'04: Proc. of the Parallel Problem Solving from Nature*, Vol. 3242. 832–842.

[91] Wei Zou, Shujian Huang, Jun Xie, Xinyu Dai, and Jiajun Chen. 2019. A Reinforced Generation of Adversarial Samples for Neural Machine Translation. *CoRR* (2019).

[92] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *ICLR'21: Proc. of the 9th International Conference on Learning Representations*.

Received 2023-09-28; accepted 2024-04-16