A Survey on Robustness Attacks for Deep Code Models

Yubin Qu[1,2,3], Song Huang[1*], Yongming Yao[1]

[1*]College of Command and Control Engineering, Army Engineering University of PLA, Nanjing, 210001, China.
[2]School of Information Engineering, Jiangsu College of Engineering and Technology, Nantong, 226001, China.
[3]Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, 541004, China.

*Corresponding author(s). E-mail(s): huangsong@aeu.edu.cn;
Contributing authors: quyubin@hotmail.com; yym879@hotmail.com;

Abstract

With the widespread application of deep learning in software engineering, deep code models have played an important role in improving code quality and development efficiency, promoting the intelligence and industrialization of software engineering. In recent years, the fragility of deep code models has been constantly exposed, with various attack methods emerging against deep code models and robustness attacks being a new attack paradigm. Adversarial samples after model deployment are generated to evade the predictions of deep code models, making robustness attacks a hot research direction. Therefore, to provide a comprehensive survey of robustness attacks on deep code models and their implications, this paper comprehensively analyzes the robustness attack methods in deep code models. Firstly, it analyzes the differences between robustness attacks and other attack paradigms, defines basic attack methods and processes, and then summarizes robustness attacks' threat model, evaluation metrics, attack settings, etc. Furthermore, existing attack methods are classified from multiple dimensions, such as attacker knowledge and attack scenarios. In addition, common tasks, datasets, and deep learning models in robustness attack research are also summarized, introducing beneficial applications of robustness attacks in data augmentation, adversarial training, etc., and finally, looking forward to future key research directions.

Keywords: Deep neural networks, adversarial examples, code models, robustness attacks

1 Introduction

With the rise of the AI heatwave based on deep learning, a new round of AI-empowered source code processing model research has been triggered, and AI technology has significantly empowered source code processing tasks in software engineering, leading to a wave of large-scale language models tailored for code understanding and processing, which we call DSCM(deep source code model). Deep learning has achieved success in various code processing tasks, with popular applications including code clone detection [1], code translation [2], vulnerability detection [3][4][5][6], vulnerability patching [7][8][9], code summarization [10][11][12] [13][14], method name prediction [15], Code Authorship Identification [16], code completion [17][18][19], self-admitted technical debt detection [20] etc. These applications demonstrate that deep neural networks possess strong code semantic understanding capabilities. They can learn features from the abstract structure and patterns of code and apply them to various code analysis and transformation tasks, bringing new opportunities to software development and maintenance. The rapid growth of DSCM is attributed to three key factors [21]: open-source data platforms represented by GitHub provide a large amount of available data for DSCM, the advancement of deep learning algorithms, and the active participation and collaboration of the open-source community. DSCM is rapidly advancing in all aspects of software engineering and has achieved remarkable success in various fields. Recently proposed models like Code Llama [22] and StarCoder [23] lead in code generation tasks. These powerful models have quickly evolved from experimental prototypes to practical tools, integrated into the daily workflow of software developers worldwide. GitHub Copilot [24] has been packaged as an extension for Visual Studio Code, with over 400,000 subscribers within just one month of its official release, demonstrating its popularity. As of September 2022, this momentum continues, with over 1.2 million developers participating in the Copilot technical preview in its first year. Based on large language model technology, a series of tools supported by different LLM4Code have been developed and deployed, including Amazon CodeWhisperer [25] and CodeGeeX [26]. These tools are more than just code completion aids; they can perform complex tasks such as code summarization and question answering. Some of these technologies have been further developed into industrial solutions to improve software development efficiency, such as the code completion tool pack TabNine [27] and aiXcoder [28]. So far, DSCM has achieved SOTA performance in accuracy or F1 score in many downstream software engineering tasks.

Deep learning (DL) models excel in source code processing tasks, achieving high accuracy (F1-score et al.). However, a crucial aspect often neglected is their robustness. A robust model makes consistent predictions even for slightly altered code snippets. This is critical because real-world code exhibits inherent diversity:

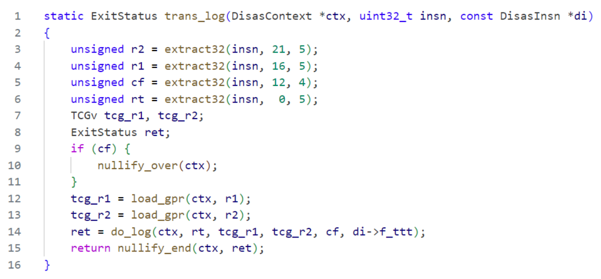
1. Coding Style Variations: Different developers write the same functionality in unique ways.
2. Developer Inconsistency: Developers may write code differently based on context.
3. Library Choices: The same functionality can be achieved using various libraries, leading to code variations.

A robust DL model should consistently classify code with the same underlying meaning for tasks like vulnerability detection, even if the surface appearance differs slightly. Inconsistent predictions can hinder real-world adoption. Robustness attacks further highlight the importance of robustness. Attackers can intentionally modify code to bypass DL models, potentially causing system failure. Such attacks can be catastrophic in security-critical tasks (malware detection, defect detection).

Studying robustness attacks on deep code models is particularly crucial as they play vital roles in critical software engineering tasks, where any deficiency could have severe consequences. For example,

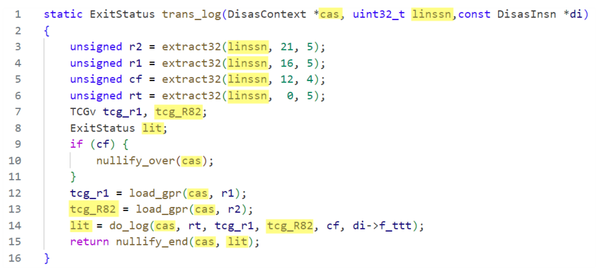
1. Vulnerability Detection: Deep code models are widely used to detect security vulnerabilities in code. If attackers can craft adversarial samples to bypass the detection models, vulnerable code may be mistakenly evaluated as secure, posing serious security risks.
2. Code Generation: Large language models like GitHub Copilot are widely adopted to assist code generation. If these models lack robustness, they may generate code with security vulnerabilities or errors, affecting the security and reliability of the entire software system.
3. Other Tasks: Deep code models are also applied to software engineering tasks such as code summarization and clone detection. Lack of robustness may lead to inconsistent predictions for slight code variants, reducing the efficiency of software development and maintenance.
4. Real-world Impact: Lack of robustness may lead to the potential severe impacts on software supply chain security, end-user privacy, and economic losses to enterprises.

In recent years, researchers in the academic community have raised concerns about the robustness of DSCM. Researchers [29] have shown that code models like code2vec [30] and code2seq [31] may produce different results for two code snippets that share the same operational semantics, with one being generated by renaming some variables in another code snippet. The modified code snippets are referred to as "adversarial samples," with the targeted model being called the "victim model" [32]. Researchers [29][33][34][35][36][37][38][32] have conducted studies and identified issues of robustness in deep source code processing models. These studies generate adversarial code snippets by renaming code variable names or transforming code segments into synonyms, ensuring that the generated adversarial code snippets meet the requirement of **syntactic validity**. Yang et al. [32] introduced ALERT, a black-box attack considering natural semantics when generating adversarial code samples. As shown in Figure 1, the local variable name *ctx* is renamed to *cas*. This adversarial attack method fully considers the **syntactic validity**, **generative efficiency**, and **naturalness** of the generated adversarial code snippets, demonstrating strong attack practicality. Deep learning-based vulnerability detection models aim to identify vulnerable code snippets. Malicious users may write code snippets vulnerable to attacks and do not want them to be identified. In security-related tasks such as malicious software or vulnerability detection, attackers may bypass vulnerability detection models by sending carefully designed code variants that contain vulnerabilities to downstream processing modules, leading to fatal errors in the entire system. In particular, the higher the



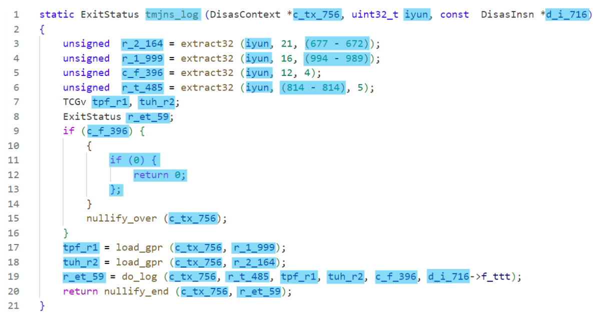
```c
static ExitStatus trans_log(DisasContext *ctx, uint32_t insn, const DisasInsn *di)
{
    unsigned r2 = extract32(insn, 21, 5);
    unsigned r1 = extract32(insn, 16, 5);
    unsigned cf = extract32(insn, 12, 4);
    unsigned rt = extract32(insn,  0, 5);
    TCGv tcg_r1, tcg_r2;
    ExitStatus ret;
    if (cf) {
        nullify_over(ctx);
    }
    tcg_r1 = load_gpr(ctx, r1);
    tcg_r2 = load_gpr(ctx, r2);
    ret = do_log(ctx, rt, tcg_r1, tcg_r2, cf, di->f_ttt);
    return nullify_end(ctx, ret);
}
```

(a) The original code snippet correctly classified by the CodeBERT model.



```c
static ExitStatus trans_log(DisasContext *cas, uint32_t linssn,const DisasInsn *di)
{
    unsigned r2 = extract32(linssn, 21, 5);
    unsigned r1 = extract32(linssn, 16, 5);
    unsigned cf = extract32(linssn, 12, 4);
    unsigned rt = extract32(linssn,  0, 5);
    TCGv tcg_r1, tcg_R82;
    ExitStatus lit;
    if (cf) {
        nullify_over(cas);
    }
    tcg_r1 = load_gpr(cas, r1);
    tcg_R82 = load_gpr(cas, r2);
    lit = do_log(cas, rt, tcg_r1, tcg_R82, cf, di->f_ttt);
    return nullify_end(cas, lit);
}
```

(b) Adversarial code by replacing variable names.



```c
static ExitStatus tmjns_log (DisasContext *c_tx_756, uint32_t iyun, const  DisasInsn *d_i_716)
{
    unsigned  r_2_164 = extract32 (iyun, 21, (677 - 672));
    unsigned  r_1_999 = extract32 (iyun, 16, (994 - 989));
    unsigned  c_f_396 = extract32 (iyun, 12, 4);
    unsigned  r_t_485 = extract32 (iyun, (814 - 814), 5);
    TCGv tpf_r1, tuh_r2;
    ExitStatus r_et_59;
    if (c_f_396) {
        {
            if (0) {
                return 0;
            };
        }
        nullify_over (c_tx_756);
    }
    tpf_r1 = load_gpr (c_tx_756, r_1_999);
    tuh_r2 = load_gpr (c_tx_756, r_2_164);
    r_et_59 = do_log (c_tx_756, r_t_485, tpf_r1, tuh_r2, c_f_396, d_i_716->f_ttt);
    return nullify_end (c_tx_756, r_et_59);
}
```

(c) Adversarial code by replacing variable names and implementing synonym substitutions in the code structure.

Fig. 1 Adversarial Code Generation Comparison

success rate of robustness attacks, the greater the chance of vulnerable code snippets evading vulnerability detection models, posing higher security risks to software quality assurance. Apart from security threats in code classification tasks, recent language models (LLM4Codes) have been shown to generate code with security issues [39][40][41][42]. An evaluation in [43] found that 40% of the programs generated by Copilot in various security scenarios contained dangerous vulnerabilities. StarCoder's repeated evaluations revealed that other state-of-the-art LLM4Codes [44][45] exhibited similarly concerning security levels as Copilot.

This trend indicates that AI security issues are becoming more relevant to real-life scenarios, and attacks on deep learning models are increasingly threatening, drawing the attention of researchers.

Although DSCM is very popular in the intelligent software engineering community, there has not been a comprehensive review article to organize and summarize the relevant research on DSCM robustness attacks. It is necessary to take a comprehensive look at the work in this research field to help subsequent researchers and practitioners have a macroscopic understanding of these methods for a comprehensive review and examination of the effectiveness and impact of robustness attack methods on DSCM. This article fills this gap by providing a comprehensive and integrated overview of robustness attacks on DSCM, which has been prompted by the sharp increase in attention to this topic. This review will help researchers and practitioners interested in attacking DSCM.

The main contributions of this review are:

1. As far as we know, we have conducted a comprehensive review and research on the robustness attacks of DSCM and have proposed different classification schemes to organize the reviewed literature;
2. Providing all relevant information, making the review itself a self-contained entity, so even readers with limited understanding of DSCM can easily comprehend;
3. Discussing unresolved issues and identifying possible research directions to establish a more robust DSCM.

The remaining part of this paper is structured as follows: Section 3 compares the attack paradigms for deep code models, including robustness attacks; Section 2 explains the definition of robustness attacks and summarizes relevant basic knowledge; Section 4 analyzes and classifies different robustness attack methods; Section 5 summarizes commonly used datasets and deep learning models for robustness attacks; Section 6 briefly discusses the beneficial applications of robustness attacks; Section 8 outlines future key research directions from various perspectives; Section 9 concludes the article and gives the final remarks.

2 Threat Model

This section briefly introduces the key components of robustness attacks and their countermeasures. We aim to help readers understand the main elements of related works on robustness attacks and their corresponding countermeasures. The threat models for attacks encompass the Attacker's Objective, Attacker's Knowledge, and Attacker's Capability [46]. This section summarizes and synthesizes the threat models, attack scenarios, and other contents in existing robustness attack research, mainly referring to the following literature [34][47][48][37][15][49][50][29][32][51].To define the main terminology, we address the following questions:

- *Adversary's Goal*: What is the attacker's objective or purpose? Do they intend to mislead the classifier's decision on a single sample or influence its overall performance?
- *Adversary's Knowledge*: What information is accessible to the attacker? Are they aware of the classifier's structure, parameters, or the training set used for classifier training?

- *Victim Models*: What types of deep learning models are adversaries typically interested in attacking? Why do adversaries target these specific models?
- *Security Evaluation*: How can we evaluate the security of a victim model when faced with adversarial examples? What is the relationship and difference between these security metrics and other model performance measures, such as accuracy or risk?

2.1 Adversary's Goal

In robustness attacks, classifiers perform well on legitimate samples but struggle with crafted fake samples. The attackers lack control over the classifier or its parameters, focusing instead on generating deceptive examples to bypass detection.

2.2 Adversary's Knowledge

Based on the attacker's knowledge of threat models, attack scenarios can be broadly categorized into two types:

- **White-Box Attack.** In a white-box setting [34][47][48][37], the adversary has access to all information of the target neural network, including its architecture, parameters, gradients, etc. The adversary can fully utilize the network information to craft adversarial examples carefully. White-box attacks have been extensively studied because disclosing model architecture and parameters helps understand the weaknesses of DNN models clearly and can be analyzed mathematically.
- **Black-Box Attack.** In a black-box attack setting [15][49][50][29][32][51], adversaries cannot access the inner configuration of DNN models. They can only input data and check the model's outputs. Typically, attackers exploit the input-output relationship of the model by inputting samples and analyzing the output to identify weaknesses. Black-box attacks are commonly used in real-world applications since designers often keep model parameters private for proprietary reasons.

White-Box Attack scenarios allow attackers to have full knowledge, making it easier to analyze the security properties of the model itself; Black-Box Attack scenarios give attackers strict knowledge conditions, focusing on applied attack research in practical scenarios.

2.3 Adversary's Target

The attacker aims to force the model to make incorrect predictions or output specific results. Both Targeted Attacks and Non-Targeted Attacks are ways of attacking a model. In a targeted attack, the attacker's goal is to ensure that the model incorrectly predicts a specific wrong category y_{target}, which means the attacker has a clear wrong output goal and strives to make the model produce that specific output. In a non-targeted attack, the attacker's goal is to make the model predict incorrectly without caring about the specific wrong category. The attacker aims to perturb the input so that the model's output does not match the true label y without needing to achieve a specific wrong category. Furthermore, in a targeted attack, the aim is to find a perturbation δ such that $M(x+\delta) = y_{\text{target}}$, meaning the model's prediction matches the target label specified by the attacker, while in a non-targeted attack, the aim is

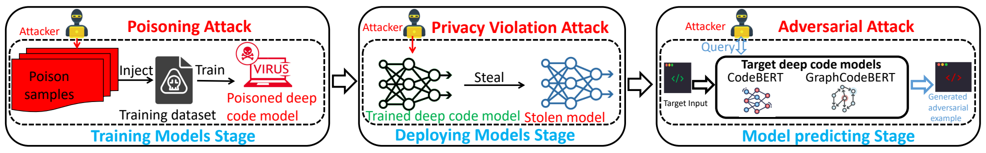


Fig. 2 Three types of attacks targeting DSCM.

to find a perturbation δ such that $M(x+\delta) \neq y$, indicating the model's prediction does not match the original true label. Both attacks have a constraint $\|\delta\|_p \leqslant \epsilon$ to ensure the generated perturbation is not too large, keeping the modified input x' in some sense close to the original input x. Finally, more intricate strategies are needed in a targeted attack to ensure the model's output becomes a specific wrong category, which may require more precise perturbation calculations. In a non-targeted attack, the strategy is relatively simple—just causing the model to predict incorrectly without precisely controlling the specific category of the wrong prediction.

2.4 Victim Models

We summarize deep learning models vulnerable to adversarial examples, focusing on studies of adversarial examples for Deep Neural Networks (DNNs). The end-to-end architecture of DNNs makes it simple for adversaries to exploit weaknesses and create high-quality deceptive inputs (adversarial examples). Detailed information about Victim Models will be briefly summarized in Section 5.

Using deep learning-based vulnerability detection models as an example, we briefly analyze the reasons why these Victim Models are vulnerable to robustness attacks:

- These models have limitations in capturing the complex logic and structure of code, potentially leading to misclassifications or incorrect predictions.
- They overly rely on superficial features or patterns, making the models susceptible to carefully crafted adversarial examples.
- They lack robustness and interpretability, making it difficult to understand the decision-making process and identify potential vulnerabilities.
- They are susceptible to distribution shifts and out-of-distribution examples, which could lead to unexpected behavior or failures.

Through sorting out and summarizing existing research, this section systematically describes the threat models, attack scenarios, etc., of robustness attacks, laying the foundation for developing subsequent content. It should be noted that the content of this section is mainly based on published literature and strives to objectively reflect the current state of research in this field.

3 Robustness Attack Paradigm in Deep Code Models

As shown in Figure2, adversarial learning currently faces three types of security threats: Poisoning Attacks, Robustness attacks(also called Adversarial attacks), and

Privacy Violation Attacks. Robustness attacks mainly occur during the inference stage of the model. For a model that has already been trained, attackers hope to construct a sample that can deceive the model without modifying or destroying the existing model.

Poisoning Attacks involve maliciously injecting a few carefully designed poisoning samples into the training dataset during the model training phase. This causes the model to learn undesirable behavior patterns during training, resulting in predictions favorable to the attacker during the inference phase. This attack targets the training dataset to influence the model's fundamental learning capability. Common poisoning attack methods include:

1. Label Flip Attack: Maliciously modifying the labels of some training samples.
2. Sample Injection Attack: Injecting carefully designed poisoning samples into the training set.
3. Logic Backdoor Attack: Causing the model to learn specific backdoor patterns and exhibit malicious behavior when a backdoor trigger is detected.

Unlike robustness attacks, poisoning attacks occur during the model training phase, while robustness attacks involve perturbing input samples during the model inference phase to achieve adversarial effects. However, both aim to mislead the model's predictions. Poisoning attacks are generally more persistent and stealthy, as the injected poisoning samples will influence the entire training process. In contrast, robustness attacks require generating adversarial perturbations for each input sample. On the other hand, poisoning attacks typically require some level of access to the training dataset, while robustness attacks do not. Overall, poisoning and robustness attacks are two different attack strategies, targeting the model training and inference phases, respectively, posing threats to model security from different perspectives.

Privacy Violation Attacks aim to extract private information from deep code models, such as sensitive data in source code or private information in the model's training data. These attacks primarily exploit the model's ability to memorize input data during the learning process and use specially designed input samples to probe and reconstruct private information. Common methods include:

1. Membership Inference Attack: Inferring whether a given data sample was used to train the model.
2. Model Extraction Attack: Reconstructing an approximate model replica to steal model knowledge.
3. Data Reconstruction Attack: Recovering partial or complete training data from the model's output.

Unlike robustness attacks, the goal of Privacy Violation Attacks is not to deceive or mislead the model's predictions but rather to steal the private information embedded in the model. Robustness attacks focus on generating adversarial samples to attack the model's prediction capability, differing in attack objectives and methods. However, they share some similarities, such as exploiting model vulnerabilities and attempting to bypass defense mechanisms. Therefore, enhancing model robustness and privacy protection capabilities is equally important, requiring careful consideration of various attack methods and the adoption of comprehensive defense measures.

Compared to other attacks, the threat of robustness attacks mainly lies in the following points:

1. Complexity: Adversarial sample attacks mainly focus on the vulnerability of models to adversarial samples during the model inference stage.
2. Stealthiness: Compared to benign samples, adversarial samples that maintain semantic consistency can evade detection by deep learning models, showing the stealthiness of the attack.
3. Practicality: Data poisoning attacks aim to make models make incorrect predictions when encountering samples with triggers. However, in practical scenarios, there is a high requirement for security permissions to poison the training dataset, limiting the application scope of backdoor attacks compared to black-box robustness attacks.

Robustness attacks refer to attackers modifying test samples to mislead the target model. Robustness attacks are the most common type among the three types of attacks. Robustness attacks can be targeted attacks (e.g., modifying samples to cause the model to misclassify them into a specified class) or nontargeted attacks (e.g., modifying samples to mislead the model). Traditionally, robustness attacks emphasize modifying samples to cause the model to make incorrect predictions while ensuring that the modified samples retain their original functionality. In recent years, the focus of robustness attacks has shifted toward further limiting the "perturbations" applied to the modified samples.

The pseudocode for a generic robustness attack process for deep code models is shown in Algorithem 1.

Algorithm 1 Generic Robustness Attack Process for Deep Code Models

```
Require: Original code sample $x$, Target deep code model $f$, Perturbation budget $\epsilon$, Number of iterations $N$
Ensure: Adversarial code sample $x_{adv}$
 1: Initialize $x_{adv} \leftarrow x$
 2: for $i = 1$ to $N$ do
 3:     Calculate the gradient of the loss function for $x_{adv}$:
 4:     $\nabla_{x_{adv}} \mathcal{L}(f(x_{adv}), y)$
 5:     Generate perturbation $\delta$ based on the gradient information
 6:     Ensure the perturbed code sample remains valid and within the perturbation budget:
 7:     $x_{adv} \leftarrow \text{clip}(x_{adv} + \delta, x - \epsilon, x + \epsilon)$
 8:     Evaluate the attack effect using the adversarial code sample $x_{adv}$
 9:     if Attack is successful then
10:         break
11:     end if
12: end for
13: return $x_{adv}$
```

The input includes the original code sample x, the target deep code model f, the perturbation budget ϵ, and the number of iterations N. Initialize the adversarial code sample x_{adv} as the original code sample x. Iterate for N steps or until the attack is successful:

- Calculate the gradient of the loss function for the current adversarial code sample x_{adv}.
- Generate a perturbation δ based on the gradient information.
- Ensure that the perturbed code sample remains valid and within the perturbation budget by clipping the values.
- Evaluate the attack effect using the adversarial code sample x_{adv}.
- If the attack is successful, break the loop.

Return the final adversarial code sample x_{adv}. This overviews the common steps in a robustness attack process for deep code models. The specific implementation details, such as the gradient calculation and perturbation generation methods, may vary depending on the chosen attack algorithm and the characteristics of the target deep code model.

For software engineering, one scenario of robustness attacks targeting deep code models could involve a development team using a deep learning-based code autocompletion tool, which recommends the next line(s) of code based on the current context. An attacker discovers that the model is susceptible to specific comments or variable naming patterns. They craft a seemingly normal code snippet but embed adversarial variable names and code structures to deceive the model into generating code recommendations with security vulnerabilities. In this case, when a developer uses the tool and unintentionally accepts these recommendations, they could inadvertently introduce security flaws or backdoors into their applications.

3.1 Formal definition of robustness attacks

Assuming we have a trained AI model, we can describe the given model as a function $f_\theta : \mathcal{C} \to \mathcal{Y}$, where $\mathcal{C}$ is the set of all code snippets and $\mathcal{Y}$ is the set of labels. For a code snippet $c \in \mathcal{C}$, the trained model predicts it as $y \in \mathcal{Y}$, meaning $f_\theta(c) = y$, with $y_{bad} \in \mathcal{Y}$ representing the chosen adversarial label. Let $\Delta(c)$ be the effective modification set of code snippet c, and let $\delta(c)$ be the new input obtained by applying modification $\delta : \mathcal{C} \to \mathcal{C}$ to c, such that $\delta \in \Delta(c)$. The perturbation for code snippets is defined as follows: $\Delta(c) = \{\delta_{v \to v'} \mid \forall c \in \mathcal{C} : \delta_{v \to v'}(c) = c_{v \to v'}\}$, where $c_{v \to v'}$ is the code snippet obtained after transforming code structure v into $\to v'$ through operations like renaming code variables and code structure synonym conversion. For example, by perturbing vulnerable code snippets using operations like variable renaming and code structure synonym conversion, the adversarial code generation process is transformed into a combination optimization problem of a vector of code synonym conversion operators. As shown in Figure 1(c), the local variable name *ctx* is renamed to *c_tx_*756, and a piece of dead code is inserted into the selection structure if, altering the syntax tree structure of the function.

Define $Var(c)$ as the set of all code synonym conversion operations present in c. The attacker's goal is to select a set of code synonym conversion operations $v \in Var(c)$ and an alternative name v', such that transforming v into v' through synonym conversion can result in the model predicting the adversarial label. The technical challenge lies in finding a set of code synonym conversion operations v' such that $f_\theta\left(\delta_{v\to v'}\left(c\right)\right) = y_{bad}$.

Constraints for adversarial samples in source code:

1. **Syntactic Validity**, unlike in the image domain, DSCM deal with code snippets. For source code, robustness attacks and robustness analysis become more challenging due to various constraints and requirements. Existing robustness analysis in image or speech domains is primarily based on distance in Euclidean space. At the same time, the program source code is discrete and strictly adheres to formal grammar (e.g., context-free grammar), making robustness measurements difficult to define using distance in Euclidean space. Only syntactically valid code snippets can be used for robustness analysis and estimation. This characteristic of adversarial code snippets is known as **Syntactic Validity**, meaning the generated code (adversarial example) must be compilable (i.e., syntactically valid) and retain the semantics of the original code snippet.
2. **Generation Efficiency**, robustness attacks must quickly find adversarial code snippets in a vast space under complex constraints.
3. **Naturalness**, meaning adversarial code snippets generated by adversarial attack methods should be understandable and natural to programmers.
4. **Attack Success Rate**, robustness attacks are typically used to estimate the robustness of deep learning models in worst-case scenarios, where stronger adversarial attack techniques can more accurately approximate real robustness attacks.
5. **Semantic Equivalent Transformation**, adversarial samples should maintain semantic consistency with the source code.
6. **Minimization of Perturbation**, in addition to semantic equivalence, it is also required that the pre and post-perturbation programs be as similar as possible, i.e., $c \approx \delta\left(c\right)$. While this constraint is clearly defined formally and intuitively in continuous domains (e.g., images and audio[52] [53]), it does not hold in discrete domains (such as programs). The main reason is that every change in the code is noticeable. A series of perturbations $\delta_1, \delta_2, ..., \delta_k$ can be constructed and applied to a given code snippet: $\delta_1 \circ \delta_2 ... \circ \delta_k\left(c\right)$.

3.2 Security Evaluation Metrics

3.2.1 Attack Success Rate

Attack Success Rate (ASR) is widely used as a metric for measuring robustness against attacks[48][33][54][15][49][16][55][56][57][58], which was initially proposed by Yefet et al. [34]. ASR is defined as the percentage of output tokens misclassified by the model on the perturbed input but correctly predicted on the unperturbed input for each token j in the expected output of sample i. The higher the ASR, the better the attack.

$$\mathrm{ASR} = \frac{\sum_{i,j} 1\left(\boldsymbol{\theta}\left(\mathbf{x}'_i\right) \neq y_{ij}\right)}{\sum_{i,j} 1\left(\boldsymbol{\theta}\left(\mathbf{x}_i\right) = y_{ij}\right)} \tag{1}$$

Jha et al. [36] used the following metric to evaluate the effectiveness of the generated adversarial code:

- Queries: In a black-box scenario, the attacker can query the target model to observe changes in the output logits. A lower average query count per sample indicates a more efficient attacker.
- Perturbation: The average number of tokens modified to create an adversarial code. A lower value indicates a less noticeable attack.

Queries are also referenced by [55]. Attack Success Rate (ASR), Queries, and Perturbation are commonly used evaluation metrics for robustness attacks, each with strengths and limitations. ASR intuitively reflects the effectiveness of the attack method and applies to various attack scenarios and methods, making it easy to understand and compare. However, it does not consider the efficiency and cost of the attack process and lacks measurement of the quality of perturbed samples. Queries, on the other hand, measure the efficiency of the attack process, with fewer queries indicating lower attack costs. It is particularly suitable for black-box attack scenarios. Nevertheless, the number of queries is related to factors such as model structure and dataset, making it difficult to directly compare different methods, and evaluating the attack effect is not comprehensive enough.

Perturbation measures the similarity between adversarial examples and original samples, with smaller perturbations indicating stronger attack imperceptibility. It has various calculation methods, such as the $L0$ norm and the $L2$ norm. However, it lacks a unified measurement standard, and different norms have large differences in calculation results. Moreover, it is difficult to quantitatively evaluate the perception of human observers. Combining specific attack scenarios and purposes is necessary to select appropriate metric combinations for comprehensive evaluation. Meanwhile, exploring more effective and unified evaluation methods in future work is essential to address the limitations and challenges associated with the current metrics.

3.2.2 Metrics Associated with Downstream Tasks

Various evaluation metrics are used for different software engineering tasks. Srikant et al. [33] replicated results from [59] using the F1-score metric. They emphasize the reproducibility of their results. CodeBLEU $_q$ [36] [55] is utilized to assess the consistency of adversarial code. Zhang et al. [60] employ Precision (P), Recall (R), and F1-Measure (F_1) Score for ML-based detector performance. Rabin et al. [15] use traditional sub-token metrics (precision, recall, F1-score) for method name prediction. Zhou et al. [51] apply standard metrics like BLEU, METEOR, and ROUGE-L for code generation quality assessment. Zhang et al. [37] use macro precision, recall, F1-score, and macro F1-score for source code classification evaluation. Yang et al. [49] consider BLEU and CodeBLEU for code generation tasks. Pour et al. [35] utilize F1-score for classification accuracy, ROUGE for code captioning, and BLEU for code document generation.

3.3 Arms Race in Robustness Attack and Defense

The robustness attack research on deep code models often presents a game process of tit-for-tat and confrontation between attackers and defenders. Attackers aim to find and exploit the model's vulnerabilities, generate effective adversarial samples, and cause the model to produce errors or failures in practical applications. Defenders, on the other hand, are committed to enhancing the model's ability to resist attacks, improving the robustness and security of the model through algorithm improvements, model reinforcement, and other means. Both sides constantly upgrade and iterate in the process of attack and defense, forming a dynamic arms race situation.

Specifically, attackers typically employ the following strategies: 1) carefully designing perturbations to mislead the model to make wrong judgments to the greatest extent while maintaining semantic equivalence; 2) using new adversarial sample generation techniques, such as gradient-based methods, optimization-based methods, etc., to continuously improve the effectiveness and efficiency of attacks; 3) exploring and excavating the weaknesses of model structure and training methods to discover new attack surfaces and breakthroughs. Defenders, on the other hand, need to take corresponding countermeasures: 1) introducing adversarial samples in the model training process to improve the generalization ability and robustness of the model; 2) developing more powerful and specialized defense algorithms to detect and filter out malicious adversarial samples promptly; 3) adopting targeted reinforcement measures in model design and implementation, such as using more complex network structures, adding robustness regularization terms, etc., to reduce the vulnerability of the model.

Take the automatic code translation task as an example. Yang et al. [61] proposed that pre-trained models (PTMs) have shown great potential in automatic code translation. However, the vulnerability of these models in translation tasks, especially in terms of syntax, has not been widely investigated. They introduced a new approach, **CoTR**, to evaluate and improve the syntactic adversarial robustness of PTMs in code translation. **CoTR** consists of two components: CoTR-A and CoTR-D. CoTR-A generates adversarial examples by transforming programs, while CoTR-D proposes a sampling data augmentation method based on semantic distance and adversarial training methods to improve the model's robustness and generalization ability. The effectiveness of **CoTR** was evaluated through experiments on a real-world Java to Python dataset. The results show that CoTR-A significantly reduces the performance of existing PTMs, while CoTR-D effectively improves the robustness of PTMs. This reflects the progress and evolution of robustness research driven by the attack-defense game.

From a higher perspective, the arms race of robustness attack and defense has had a profound impact on the development and application deployment of deep code model technology. On the one hand, driven by the attack-defense confrontation, researchers continue to innovate algorithms and models, designing solutions with better performance and stronger robustness, promoting the technological progress of deep code models. On the other hand, the attack-defense arms race also highlights the security risks and challenges faced by models in practical applications. Developers need to focus on the robustness and security of the model during the deployment and update process, comprehensively balance the performance and security trade-offs, and establish a continuous security testing and monitoring mechanism to deal with potential attack threats. This puts higher requirements for applying deep code models in practical software development tasks.

4 Generating Adversarial Examples

This section introduces the main methods for generating adversarial examples in DSCM, summarizing the current research progress on robustness attacks in DSCM. We go through some methods to generate adversarial code examples for robustness attacks and analyze and classify them according to the characteristics of each robustness attack method.

4.1 White-box Attacks

In a white-box attack scenario, when the classifier C (model F, also DNNs) and the victim sample (x, y) are provided to the attacker, the objective is to generate a fake image x' or an adversarial source code that closely resembles the original image x but can deceive the classifier C into providing incorrect predictions. This can be described as:

$$\begin{aligned}&\text{find } x' \text{ such that } ||x' - x|| \leqslant \epsilon\\&\text{with the condition } C(x') = t \neq y\end{aligned}$$

The formula describes the objective and constraints of a white-box attack, where: x represents the original input sample, x' represents the generated adversarial sample, $C(x')$ represents the model's prediction for x', y represents the model's correct prediction for the original sample x, t represents the attacker's desired incorrect prediction for x', $|x' - x|$ represents the distortion size between x' and x, controlled within a certain range ϵ.

The attack aims to find an adversarial sample x' such that: x' is sufficiently close to the original sample x, with the distortion size $|x' - x|$ controlled within the range ϵ, ensuring similarity of the adversarial sample. The model's prediction for x', $C(x')$, equals the attack target t and does not equal the original correct prediction y. Therefore, this is a constrained optimization problem: generating adversarial samples that can mislead the model under the constraint of distortion size. This white-box attack is typically conducted when the model's internal details, such as weights and structure, are known. Consequently, the gradient information of the adversarial perturbation can be calculated in a targeted manner, generating stronger adversarial samples. The key to a successful attack is striking a balance between the adversarial perturbation size and the adversarial effect, ensuring that the perturbation is small enough while still capable of deceiving the model. This is crucial for improving model robustness.

4.1.1 Yefet's DAMP Attack

Yefet et al. [34] proposed a method to leverage adversarial example attacks against DSCM. The main idea of their method is to introduce small perturbations that do not change the semantics of the program, thereby creating an adversarial example that forces the given DSCMto make an erroneous prediction specified by the adversary. They introduced a new technique called Discrete Adversarial Manipulation of Programs (DAMP) to find such perturbations. DAMP works by deriving the desired prediction result from the model's input while keeping the model weights fixed and then slightly modifying the input code based on the gradient.

4.1.2 Bielik's Attack

Bielik et al. [47] proposed a robust training strategy that trains the model to abstain from making decisions when uncertain whether the input program has undergone adversarial perturbations. They converted the program into a graph representation and defined the model as a graph neural network that propagates and aggregates messages along the graph edges. Since graph neural network dependencies were based on the graph's structure (i.e., its edges), they framed refining the representation as an optimization problem. They aimed to learn an abstraction function that removes some edges from the graph and presented an efficient solution to the optimization problem by transforming it into an integer linear program.

4.1.3 Gao's Attack

Gao et al. [48] studied discrete adversarial examples created through program transformations that preserve the semantics of the original input and proposed a novel general method to attack a broad range of code models efficiently. From the defensive perspective, their main contribution is providing a theoretical foundation for adversarial training—the most successful algorithm for training robust classifiers—to defend against discrete robustness attacks.

4.1.4 Zhang's CARROT Attack

Zhang et al. [37] proposed a framework called CARROT for detecting, measuring, and enhancing the robustness of deep learning models for source code processing. They first introduced an optimization-based attack technique called CARROTA to efficiently generate effective adversarial source code examples. They defined robustness metrics and proposed a robustness measurement toolkit called CARROTM, which approximates the worst-case performance under allowed perturbations. Further, they proposed adversarial training (CARROTT) and attack techniques to improve the robustness of deep learning models.

4.1.5 Ramakrishnan's Attack

Ramakrishnan et al. [59] investigated the robustness issue of deep source code models, proposing that deep code models should be robust to semantics-preserving source code modifications. They defined a strong adversary capable of using a suite of parameterized, semantics-preserving program transformations. They also demonstrated how to perform adversarial training to learn robust models for such an adversary.

Given a set of transformations $\mathcal{T}$, a k-adversary is an oracle that finds a sequence of transformations of size k that maximizes the loss function. They used q to denote a sequence of transformations $t_1, \ldots, t_n$, and their corresponding parameters $r_1, \ldots, r_n$, where $t_i \in \mathcal{T}$ and $r_i \in \mathcal{R}$. We use $q(x)$ to denote the program $t_n(\ldots t_2(r_2, t_1(r_1, x)))$, i.e., the result of applying all transformations in q to x. Let $\mathcal{Q}^k$ denote the set of all sequences of transformations and length parameters k. Given a program x with label y, the goal of the adversary is to transform x to maximize the loss; formally, the adversary solves the following objective function:

$$\max_{q \in \mathcal{Q}^k} L(w, q(x), y) \tag{2}$$

4.1.6 Srikant's Attack

Srikant et al. [33] proposed a principled approach for fooling these learning models by subjecting computer programs to adversarial perturbations, thereby assessing their adversarial resilience. They used program obfuscation, traditionally employed to prevent reverse engineering of computer programs, as the adversarial perturbation. These perturbations modify the programs in a functionality-preserving manner but can be designed to mislead machine learning models at decision time. They provided a general adversarial program formulation that allows applying multiple obfuscation transformations to programs in any language.

Let $\mathcal{P}$ denote a *benign program* consisting of a series of n tokens $\{\mathcal{P}_i\}_{i=1}^n$ in the source code domain. Let $\mathcal{P}'$ define a *perturbed program* (for $\mathcal{P}$) created by solving the *site-selection* and *site-perturbation* problems, which utilize the vocabulary ω to find an optimal replacement. To formalize the *site-selection* problem, they introduced a vector of boolean variables $\mathbf{z} \in \{0,1\}^n$ to indicate whether or not a site is selected for perturbation. To define *site-perturbation*, they introduced a one-hot vector $\mathbf{u}_i \in \{0,1\}^{|\omega|}$ to encode the selection of a token from ω which would serve as the insert/replace token for a chosen transformation at a chosen site. Using the above formulations for *site-selection*, *site-perturbation*, and *perturbation strength*, the *perturbed program* $\mathcal{P}'$ can then be defined as

$$\mathcal{P}' = (\mathbf{1} - \mathbf{z}) \cdot \mathcal{P} + \mathbf{z} \cdot \mathbf{u}, \text{ where } \mathbf{1}^T\mathbf{z} \leqslant k, \mathbf{z} \in \{0,1\}^n, \mathbf{1}^T\mathbf{u}_i = 1, \mathbf{u}_i \in \{0,1\}^{|\omega|}, \forall i, \tag{3}$$

4.1.7 He's SVEN Attack

Large language models are increasingly being trained on large-scale code repositories and used for code generation. In recent years, large code language models (LMs) have demonstrated powerful code understanding capabilities, and attacks against LMs have just begun to emerge. LMs lack awareness of security and often generate insecure code. He et al. [42] conducted adversarial testing of LMs by formulating a new security task called controlled code generation, aiming to evaluate the security of LMs under adversarial stances and proposed a novel learning-based method called SVEN to address this task. SVEN utilizes attribute-specific continuous vectors to guide program generation towards given attributes without modifying the weights of the LM. Using carefully curated high-quality datasets, SVEN's training process optimizes these continuous vectors by applying specialized loss terms on different code regions.

4.1.8 Summary of White Box Attack Methods

Yefet et al. [34] suggest perturbing programs by replacing local variables and inserting print statements with replaceable string parameters. They use a first-order optimization method to find optimal replacements, similar to [62] and HotFlip [63]. This is an improvement over [29], which uses the Metropolis-Hastings algorithm to find the best replacements for variable names. [47] propose a robust training strategy that trains models to abstain from decisions when uncertain whether the input program has been adversarially perturbed. They consider a small space of transformations and search for them greedily. Moreover, their solution is designed to reason about limited contexts of programs (predicting variable types) and is difficult to scale to applications that require reasoning about the entire program, such as program summarization (studied in this work). Ramakrishnan et al. [59] extend the work of [34]. They experiment with more transformations and propose a standard min-max formulation for adversarially training robust models. Their inner maximizer generates adversarial programs while applying multiple transformations to a program, unlike [34]. Srikant et al. believe that optimizing only the locations can improve attack performance and propose a joint optimization problem, namely finding optimal locations and optimal transformations, which only the latter [59] (and [34]) can solve in a principled manner.

4.2 Black-Box Attacks

Black-box attacks do not require details of the neural network but have access to inputs and outputs. This type of attack often relies on heuristic methods to generate adversarial examples because the details of the DNN are a black box to the attacker in many real-world applications.

4.2.1 Rabin's Attack

Rabin et al. [15] propose that neural program models have been tested on various existing datasets, but the extent to which they generalize to unforeseen source code remains largely unknown. They compare the results of various neural program models in performing the method name prediction task on programs before and after automatic semantics-preserving transformations. Nine such neural program models are evaluated using three Java datasets of different scales and three state-of-the-art code neural network models, namely code2vec, code2seq, and GGNN. Their results show that these neural program models often fail to generalize their performance even with small semantics-preserving changes to programs; neural program models based on data and control dependencies in programs generalize better than those based solely on abstract syntax trees; as the size and diversity of the training dataset increase, the generalization ability of correct predictions made by neural program models also improves.

4.2.2 Yang's RADAR Attack

Yang et al. [49] propose studying and demonstrating the potential of leveraging method names to improve the performance of pre-trained code generation models(PCGMs) from the perspective of model robustness. They introduce a new approach called RADAR (Neural Code Generation Robustifier). RADAR consists of two parts: RADAR Attack and RADAR Defense. The former attacks PCGMs by generating adversarial method names as part of the input, which is semantically and visually similar to the original input but may mislead PCGMs into generating completely irrelevant code snippets. To combat this attack, RADAR-Defense synthesizes a new method name from the functional description and provides it to the PCGMs.

4.2.3 Quiring's Attack

Quiring et al. [50] implemented a dual attack approach blending compiler engineering and adversarial learning. They proposed semantics-preserving code alterations and detailed five types of source-to-source modifications. Additionally, they employed Monte-Carlo tree search as a universal black-box attack to combine transformations to achieve a target in the feature space.

4.2.4 Zhang's MHM Attack

Zhang et al. [29] formalize the adversarial sample generation process as a sampling problem. The problem can be decomposed into an iterative process involving three stages: (1) selecting a variable to rename, (2) selecting a replacement, and (3) deciding whether to accept replacing the variable with the selected replacement. To perturb the source code snippets within the constraints, they proposed the Metropolis-Hastings Modifier (MHM) method, which performs iterative identifier renaming based on Metropolis-Hastings (M-H) sampling. MHM takes the source code classifier under attack ($\mathcal{C}$) and a pair of correctly classified data ($(x, y) \in \mathcal{D}$) as input, and outputs a sequence of adversarial examples ($\hat{x}_1, \hat{x}_2, \cdots$). The adversarial examples must satisfy three requirements: 1) the ability to mislead the subject model, 2) freedom from compilation errors and the ability to be executable, and 3) the ability to produce the same execution results given the same input instances as the original examples. Their algorithm is a classical Markov chain Monte Carlo sampling approach. Given the stationary distribution ($\pi(x)$) and transition proposal, $\mathrm{M}-\mathrm{H}$ can generate desirable examples from $\pi(x)$. Specifically, at each iteration, a proposal to jump from x to x' is made based on the transition distribution ($Q\left(x' \mid x\right)$). This method is a type of black-box attack that randomly selects replacements for local variables and decides whether to accept or reject the replacement based on the predicted label and corresponding confidence. It more effectively selects adversarial samples using the victim model's predicted labels and corresponding confidence.

4.2.5 Yang's ALERT Attack

Yang et al.[32] suggest that while current studies on attacking code models are effective[29, 29, 35], they primarily focus on maintaining operational semantics and overlook whether adversarial samples are perceived as natural by human evaluators. The leading black-box approach, MHM [29], selects replacements randomly from variable names without considering the semantic connections between the original variables and their substitutes. The lack of emphasis on naturalness in existing attack strategies has prompted the introduction of "ALERT" (Naturalness Aware Attack).

This black-box attack method considers the natural semantics when creating adversarial code examples. Like MHM, "ALERT" also alters variable names to craft adversarial instances, but it does so in a way that preserves the naturalness of the code.

4.2.6 Zhou's ACCENT Attack

Zhou et al. [51] introduce a new method named *ACCENT* (**A**dversarial **C**ode **C**omment g**EN**era**T**or), which aims to create adversarial examples to enhance the resilience of neural networks in the context of code comment generation. They emphasize the importance of various identifiers within a code snippet and propose a technique that iteratively changes their names without compromising the syntax and semantics of the code. This approach focuses on generating adversarial examples that maintain the original functionality of the code while challenging the robustness of the neural networks used for code comment generation.

4.2.7 Liu's SCAD Attack

Liu et al. [16] investigated adversarial stylometry of source code that confuses source code author identification (SCAI) models. They proposed source code author disguise (SCAD) to automatically hide programmer identities and avoid author identification, which is more practical than previous work that required knowing the output probabilities or internal details of the target SCAI model.

4.2.8 Choi's TABS Attack

Choi et al. propose [56] that previous studies on black-box robustness attacks generate adversarial examples through simple greedy search, which is very inefficient. They propose TABS, an efficient beam search-based black-box adversarial attack method that employs beam search to find better adversarial examples and uses context semantic filtering to reduce the search space effectively.

4.2.9 Nguyen's GraphCodeAttack Attack

Nguyen et al.[64] propose that ALERT[32] and CARROT [37] are state-of-the-art techniques for robustness attacks on code models. These techniques focus on using a fixed set of hand-crafted patterns to transform the inputs of code models. ALERT [32] employs variable renaming, while CARROT [37] utilizes additional transformations, such as adding dead-code with manually-designed patterns like `while(false)` and `if(false)`. However, attacking code models using hand-crafted transformations presents certain limitations. In particular, the hand-crafted patterns may not keep pace with rapidly growing datasets, making it difficult to adequately represent the diverse range of real-world code structures. Furthermore, these patterns may have limitations in modeling complex semantic-preserving transformations, essential for generating effective adversarial examples. They proposed a novel adversarial attack framework called GraphCodeAttack. Given a target code model, GraphCodeAttack uses a set of input source codes to probe the model output and identify "discriminative" AST patterns that can influence model decisions. Then, GraphCodeAttack selects appropriate AST patterns, concretizes the selected patterns into attacks, and inserts them as dead code into the model's input program.

4.2.10 Jha's Code-attack Attack

Jha et al. [36] suggest that pre-trained programming language (PL) models (such as CodeT5, CodeBERT, GraphCodeBERT, etc.) have the potential to automate software engineering tasks involving code understanding and code generation. However, these models operate in the natural code channel, primarily focusing on human understanding of code. They lack robustness to variations in the input and may, therefore, be vulnerable to robustness attacks in the natural channel. The authors propose Code Attack, a simple yet effective black-box attack model that leverages code structure to generate valid, efficient, and imperceptible adversarial code samples. This demonstrates the vulnerability of state-of-the-art PL models under code-specific robustness attacks. Although some works focus on adversarial examples for code summarization [59][51], they do not operate in the natural channel. They also do not test the transferability to different tasks, PL models, and programming languages. The authors assume black-box access to state-of-the-art PL models for generating robustness attacks for code generation tasks, such as code translation, repair, and summarization. They employ a constrained code-specific greedy algorithm to find meaningful substitutes for vulnerable tokens, irrespective of the input programming language.

Given an input code sequence as a query, the adversary's goal is to degrade the quality of the generated output sequence by imperceptibly modifying the query in the natural channel of code. The generated output sequence can be a code snippet (code translation, code repair) or natural language text (code summarization). Formally, given a pre-trained PL model $F : X \to Y$, where X is the input space and Y is the output space, the goal of the adversary is to generate an adversarial sample $\mathcal{X}adv$ for an input sequence $\mathcal{X}$ *such that*:

$$\mathcal{X}adv = \underset{\mathcal{X}'}{\operatorname{argmax}}\, \mathcal{L}(F(\mathcal{X}'), Y), \quad \text{s.t.} \quad \mathcal{D}(\mathcal{X}', \mathcal{X}) \leqslant \epsilon, \tag{4}$$

where $\mathcal{L}$ is a loss function that measures the dissimilarity between the generated output sequence $F(\mathcal{X}')$ and the ground truth output Y, $\mathcal{D}$ is a distance metric that quantifies the similarity between the original and adversarial samples, and ϵ is a small positive constant that limits the magnitude of the perturbation.

$$F(\mathcal{X}_{adv}) \neq F(\mathcal{X}) \tag{5}$$

$$Q(F(\mathcal{X})) - Q(F(\mathcal{X}_{adv})) \geq \phi \tag{6}$$

where $Q(\cdot)$ quantifies the quality of the produced output and ϕ represents the designated decrease in quality. This is in conjunction with the previously imposed limitations on $\mathcal{X}_{adv}$. The ultimate objective is to create adversarial samples is articulated as follows:

$$\Delta_{atk} = \operatorname{argmax}_{\delta} \left[Q(F(\mathcal{X})) - Q(F(\mathcal{X}_{adv}))\right] \tag{7}$$

In the above objective function, $\mathcal{X}_{adv}$ is a minimally perturbed adversary subject to constraints on the perturbations δ. It searches for a perturbation Δ_{atk} to maximize the difference in the quality $Q(\cdot)$ of the output sequence generated from the original input code snippet $\mathcal{X}$ and that by the perturbed code snippet $\mathcal{X}_{adv}$.

4.2.11 Zhang's RNNS Attack

Zhang et al. [54] propose that pre-trained code models are primarily evaluated through in-distribution test data. The robustness of the models, i.e., their ability to handle out-of-distribution hard data, still lacks assessment. They introduce a novel search-based black-box adversarial attack guided by the behavior of pre-trained programming language models, named "Representation Nearest Neighbor Search (RNNS)," to evaluate the robustness of pre-trained PL models. Unlike other black-box robustness attacks, RNNS uses model change signals to guide the search in a variable namespace collected from real-world projects. Black-box attack methods based on variable substitution have become a valuable research avenue, with works like ALERT [32] and MHM [29] being proposed. However, these works have three main limitations: 1) The number of substitute variables is restricted and lacks diversity, reducing the maximum success rate of attacks. 2) The verification cost of substitute variables is high. 3) The generated adversarial samples have significant perturbations. They introduce a search-based black-box adversarial attack method to generate challenging adversarial samples based on a search seed vector in the variable representation space, termed **Representation Nearest Neighbor Search** (*RNNS*).*RNNS* utilizes publicly available real code datasets to build a large original substitute set, denoted as $subs_{original}$. Subsequently, leveraging previous attack results, *RNNS* predicts the search seed vector needed for the next rounds of attacks and efficiently looks for the k nearest substitutes to the seed vector from the extensive original substitute set to create $subs_{topk}$, with k significantly smaller than the original substitute set.

4.2.12 Chen's Attack

Chen et al. [57] propose program transformation methods, including identifier renaming and structural transformations, ensuring the perturbed program retains its original semantics while fooling source code processing models and changing the original prediction results. First, the important tokens are found by calculating the contribution value of each part of the program. Then, the optimal transformation is selected for each important token to generate semantic adversarial examples.

4.2.13 Pour's Attack

Pour et al. [35] note that in recent years, the practical application of deep neural networks (DNNs) in source code processing tasks (such as clone detection, code search, and comment generation) in the field of software engineering has been continuously expanding. Although there has been a lot of work recently on testing DNNs in the context of image and speech processing, progress in DNN testing in the context of source code processing has been limited so far, as source code processing presents unique features and challenges. To generate new test inputs, popular source code refactoring tools are used to generate semantically equivalent variants. To perform testing more effectively, DNN mutation testing is utilized to guide the testing direction.

A few studies in the literature propose specialized adversarial generation strategies for code embedding. For example, Rabin et al. and Ramakrishnan et al. suggest using program refactoring operators to generate adversarial examples for source codes. While their adversarial code example generator is also based on refactoring operators, they advocate adopting mutation testing-based guidance for more effective generation. Moreover, unlike prior work, they enhance model robustness by retraining them with adversarial examples, demonstrating significant improvement in various downstream tasks.

4.2.14 Tian's CODA Attack

Several adversarial example generation techniques tailored to deep code models have recently been proposed, including MHM [29], CARROT [37], and ALERT [32]. These approaches face significant challenges: the vast ingredient space defined by code transformation rules. Current methods often rely on modifying target input and using resulting model prediction changes to drive the search process, potentially leading to local optima and hindering test effectiveness. Repeated use of the target model may impact test efficiency in generating adversarial examples, as model invocation tends to be the most resource-intensive aspect during testing.

Tian et al. [38] propose a technique called **CODA** (**CO**de **D**ifference guided **A**dversarial example generation). The key idea is to leverage the code differences between the target input (i.e., the given code snippet as the model input) and a reference input (i.e., an input with small code differences but different prediction results from the target input) to guide the generation of adversarial examples. It considers both structural differences and identifier differences to preserve the original semantics. Therefore, the component search space can be greatly reduced since it consists of these two types of code differences, and the testing process can be improved by designing and guiding the corresponding equivalent structural transformations and identifier renaming transformations.

4.2.15 Na's DIP Attack

Na et al. [55] propose DIP (a Dead-code Insertion based black-box attack for Programming language models), which is a high-performance, efficient black-box attack method that uses dead code insertion to generate adversarial examples.

4.2.16 Zhang's DRLSG Attack

Zhang et al. [60] propose challenging the robustness of recent machine learning-based clone detectors through code semantics-preserving transformations. They introduce a Deep Reinforcement Learning-based Sequence Generation (DRLSG) strategy to effectively guide the search process for generating clone code snippets that can evade detection.

4.2.17 Tian's QMDP Attack

Many approaches have been used to generate adversarial examples by modifying the original inputs of DL models, such as variable renaming, equivocal expression substitution, and dead code insertion. However, these approaches do not consider the significance of code structure features. Tian et al. [58] propose the hypothesis that the structural features of the source code are non-robust, and adversarial examples can be generated by modifying the structural information of the source code. They propose a Q-Learning-based Markov Decision Process (QMDP) to perform semantically equivalent transformations of the source code structure. This approach addresses two key issues: (i) how to attack the structural information of the source code and (ii) when and where to execute what kind of transformation in the source code. The proposed QMDP framework formulates the problem of generating adversarial examples as a sequential decision-making process. The agent learns to make optimal decisions on selecting the most effective transformations to apply at each step, considering the current state of the source code and the potential impact on the model's prediction. By iteratively applying these transformations, the agent aims to generate adversarial examples that maintain the original semantics of the code while effectively fooling the target DL model.

4.2.18 Summary of Black Box Attack Methods

Black-box robustness attack methods for deep code models are summarized as follows:

1. Adversarial example generation: Black-box attack methods aim to generate adversarial examples while preserving the code's functionality and semantics.
2. Heuristic-based approaches: Black-box attacks often rely on heuristic methods to generate adversarial examples because they lack access to the model's gradients or internal structure.
3. Iterative process: Many black-box attacks employ an iterative process to generate adversarial examples. This process typically involves selecting a component (e.g., variable, expression) to modify, choosing a suitable replacement, and deciding whether to accept the modification based on the model's output.
4. Naturalness and semantics preservation: Some black-box attacks, such as ALERT and ACCENT, emphasize the importance of generating adversarial examples perceived as natural by human evaluators and maintaining the original code semantics.
5. Model robustness evaluation: Black-box attacks are used to evaluate the robustness of deep code models, particularly their ability to handle out-of-distribution or adversarial inputs
6. Transferability: Some studies investigate the transferability of adversarial examples generated by black-box attacks across different tasks, models, and programming languages.

4.3 Differences from Other Domains

[65], [50], [15], and [66] convert source code or binary code obfuscation into potential adversarial samples; however, they do not find an optimal set of transformations to deceive downstream models. Pierazzi et al. [66] studied machine learning-based robustness attacks on Android malware classification and proposed a novel formalization of adversarial machine learning evasion attacks, including a comprehensive set of constraints defining available transformations, preserved semantics, robustness to preprocessing, and plausibility. [66] focuses on binary code, while our research targets source code; additionally, we consider a threat model based on deep neural networks, whereas [66] studies robustness attacks based on machine learning (including support vector machines, etc.). Liu et al. [67] provide a stochastic optimization method to optimally obfuscate a program by maximizing its impact on an *occluded language model* (OLM). However, they do not address the adversarial robustness of programs; their formulation is only for finding the correct transformation sequence that increases the perplexity of the OLM. They use an MCMC-based search to find the optimal sequence. Program obfuscation is a common practice in software development to obscure source code or binary code to prevent humans from understanding the purpose or logic of software.

4.4 Summary of Robustness Attack Methods

Table 1 summarizes all the attack methods. White-box attack methods generate adversarial examples through optimization, while black-box attack methods primarily generate adversarial examples through heuristic algorithms. The table shows that the number of studies on black-box attacks is far greater than that on white-box attacks. Unlike robustness attacks in the domain of graphical image processing, perturbations to DSCM cannot generate new samples through gradient computation, as DSCM deals with discrete code inputs.

Furthermore, white-box and black-box attacks share the similarity of generating adversarial samples by introducing adversarial example generation operators. The most commonly used operators in adversarial example generation include variable renaming, dead code insertion, etc. Almost every black-box attack includes variable renaming, and the effectiveness of this operator has been verified in deep code models; additionally, dead code insertion is widely used in black-box attack research. Perturbing source code using these operators can ensure the consistency of source code semantics and is a conventional and effective robustness attack method in the DSCM domain. However, researchers have adopted various strategies for using these operators to generate adversarial code snippets. White-box attacks employ optimization-based strategies, while common strategies in black-box attacks include MHM sampling, heuristic algorithms (e.g., genetic algorithms), and reinforcement learning-based generation strategies.

Moreover, according to the attack target, Roubstness Attack Methods can be divided into two main categories: Targeted Attack and Non-targeted Attack. Targeted attacks aim to make the model output a specific incorrect category. Attackers must

Attacking Capability	Adversarial example generation operators
White box	Variable Renaming;Dead-Code Insertion;[34]
	AddDeadCode;RenameParameter;RenameField;RenameLocalVariable; ReplaceTrueFalse;Add function parameters;[47]
	Variable Renaming;Dead-Code Insertion;[48]
	Var renaming;Data type renaming;Func renaming;Empty stmt ins/del;Branch ins/del;Loop ins/del;[37]
	AddDeadCode;RenameParameter;InsertPrintStatement;RenameField; UnrollWhile;RenameLocalVariable;WrapTryCatch;ReplaceTrueFalse;[59]
	replace and insert transformations;[33]
	controlled text generation;[42]
Black box	Variable Renaming;Permute Statement;Unused Statement;Loop Exchange;Switch to If;Boolean Exchange;[15]
	Delete Operator;Swap Operator; Replace-vis Operator;Replace-sem Operator;[49]
	Control transformations;Declaration transformations;API transformations ;Template transformations ;Miscellaneous transformations;[50]
	Variable Renaming;[29]
	Variable Renaming;[32]
	Variable Renaming;[51]
	Trivial Transformations;Data Transformations;Control Flow Transformations;Function Transformations:Add Bogus Code;[16]
	Variable Renaming;[56]
	Generated codes based on abstract syntax trees;[64]
	Operator (character) level substitution- only an operator is inserted/replaced/deleted; Token-level substitution;[36]
	Variable Renaming;[54]
	Identifier renaming;Loop exchange;Boolean exchange;Prefix and suffix exchange;[57]
	Local Variable Renaming;Argument Renaming;Method Name Renaming;API Renaming;Local Variable Adding;Argument Adding;Add Print;For Loop Enhance;IF Loop Enhance;Return Optimal;[35]
	R1-loop;R2-branch;R3-calculation;R4-constant;[38]
	dead code insertion;[55]
	Op1-ChRename;Op2-ChFor;Op3-Chwhile;Op4-ChDo;Op5-ChifElseIF;Op6-Chif;Op7-ChSwitch;Op8-ChRelation;Op9-ChUnary; Op10-Chincrement;Op11-ChConstant;Op12-ChDefine;Op13-ChAddJunk;Op14-ChExchange;Op15-ChDelete;[60]
	Declaration statement splitting;Initialization assignment splitting;Declaration initialization syncing;Multi-variable assignment splitting;Multi-variable assignment merge;Input API transformation;Output API transformation;Self-increasing/self-decreasing expressions unfolding;Prefix and suffix operator swapping;For/while statement transformation;If statement cascading;If statement splitting;Switch/if transformation;Ternary/if transformation;[58]

Table 1 Adversarial example generation operators for black-box attacks

carefully design perturbations to guide the model in making specific wrong predictions. Non-targeted attacks aim to mislead the model to make incorrect predictions without specifying which wrong category to output. Attackers only need to perturb the input to be inconsistent with the true label. Research has shown that predicting method names is easily influenced by code transformations, resulting in uncertain method names[35][15]. In addition to robustness attacks in code generation[56][49][42] and code summarization[51][35][36][57][59][33], robustness attacks in classification tasks exist. In functionality classification[38][58][37][29] and authorship attribution tasks[50][16][55][38][32][64][54], when there is a slight perturbation in the code, the predicted results are not robust. Three binary software engineering tasks including defect prediction[38][37][48][34],clone detection[64][38][32][54][55][60], and vulnerability prediction[38][55][54][64][32], can be thought of as targeted attacks. The outcome of robustness perturbations can lead to defects not present or clones not present in the task code of deep learning models.

By comparison, it can be found that targeted attacks usually require more sophisticated perturbation strategies and more queries, while non-targeted attacks are relatively simple, focusing on disrupting the model's original judgment without precisely controlling its output. However, both attacks have confirmed the lack of robustness in code processing models. More efficient and stealthy black-box attack methods can be explored in the future.

5 Commonly Used Datasets and Models

This section summarizes the commonly used datasets and deep learning models in DSCM robustness attacks, categorized by task, as shown in Table 2. We use GNN to represent Graph Convolutional Networks and GGNN to represent Gated Graph Neural Networks in the column 'Threat models'. Please refer to the corresponding literature for neural network models represented by abbreviations that are not universal. The commonly used datasets and models include classification tasks and generation tasks, such as code comment generation and code generation. Robustness attacks on DSCM have been extensively studied in multiple tasks, including Defect Prediction, Functionality Classification, Authorship Attribution, Clone Detection, Code Generation, Code Summarization, and Vulnerability Prediction. Most studies have used public datasets with reproducible algorithms, while only a few have used private datasets.

Tasks	Datasets	Threat models
Code Captioning[35]	Java-large dataset[31]	Code2seq
Defect Prediction[38][37][48][34]	Codechef[1];CodeSearchNet[68];Java-large dataset[31]	CodeBERT;GraphCodeBERT;CodeT5; GRU; LSTM;ASTNN;LSCNN;TBCNN; CDLH;code2vecGGNN;GNN-FiLM
Functionality Classification[38][58][37][29]	The Open Judgebenchmark [69]	CodeBERT;GraphCodeBERT;CodeT5; AST-based ASTNN;LSTM;GRU;LSCNN; TBCNN;CDLH
Authorship Attribution[50][16][55][38][32][64][54]	The Google Code Jam dataset[70]	LSTM; RFC;RNN-RFC; CodeBERT;GraphCodeBERT;CodeT5
Clone Detection[64][38][32][54][55][60]	BigCloneBench[71];The Open Judge benchmark[69]	CodeBERT;GraphCodeBERT;CodeT5; ASTNN;TBCCD
Code Generation[56][49][42]	CodeSearchNet[68];CONCODE[72]; CoNaLa[73];CrossVul[74];BigVul[75]; VUDENC[76]	CodeGPT;PLBART;CodeT5;CodeGen; InCoder;SantaCoder
Code Repair[36]	The small dataset[77]	CodeT5;CodeBERT;GraphCodeBert
Code Search [35][56]	CodeSearchNet[68]	CodeBERT
Code Summarization[51][35][36][57][59][33]	Java dataset[78];Python dataset[79]; CodeSearchNet[68];Java-small dataset[31]; Java datasets[80];Py150k dataset[81]	LSTM;Transformer;GNN;CSCG;Rencos; CodeBERT;CodeT5;CodeBERT; RoBERTa;seq2seq
Code Translation[36]	Code translation datasets[2]	CodeT5;CodeBERT;GraphCodeBert
Method Name Prediction[35][15]	Java-large dataset[31]	code2vec;code2seq;GGNN
Predicting Code Types[47]	top starred projects on Github [47]	LSTM;DeepTyper;GNN;GGNN
Semantic Labeling[34][48]	Java-large dataset[31] ;C# projects[80]	code2vec;GGNN;GNN-FiLM
Vulnerability Prediction[38][55][54][64][32]	devign[82]	CodeBERT;GraphCodeBERT;CodeT5

Table 2 Commonly Used Datasets and Models

Limitations of the commonly used datasets:

- Limited dataset scale that may not cover enough code samples. For example, CodeSearchNet is relatively small and may fail to capture the diversity of programming languages.
- Existing datasets mainly focus on popular languages like Java and Python, lacking coverage for emerging or niche languages.
- Most datasets are built from open-source code on GitHub, etc., which may have quality issues that affect the dataset quality.
- There is a lack of large-scale public datasets for specific tasks like vulnerability prediction and code repair, hindering research in these areas.

6 Beneficial Applications of Robustness Attacks

Robustness attacks are not exclusively used to attack models and pose security threats. Many scholars have begun investigating the use of robustness attack methods for data augmentation, which enhances models' robustness and generalization ability to fortify deep code models.

6.1 Data Augmentation

Data augmentation refers to extending the original training set by generating adversarial examples and attempting to expose the model to more data during the training process. Data augmentation is commonly used to counter black-box attacks. Dong et al. [83] propose a data augmentation method called MIXCODE, which aims to supplement effective training data effectively. Inspired by recent advances in the computer vision domain, such as Mixup, MIXCODE generates transformed code with labels consistent with the original data using various code refactoring methods. It then adapts the Mixup technique to mix the original and transformed codes to enhance the training data.

6.2 Robust Adversarial Training

Li et al. [84] mention using semantic-preserving adversarial code embedding (SPACE) for pre-trained language models to find the worst-case semantic-preserving attacks while forcing the model to predict the correct labels in these worst-case scenarios. Experiments and analyses show that SPACE can improve the performance of code PrLMs. Zhang et al. [29] validate that adversarial training is useful for improving adversarial robustness using the adversarial samples generated by MHM. Li et al. [85], focusing on the source code authorship attribution task, propose combining data augmentation and gradient augmentation in the adversarial training phase to increase the diversity of training examples and generate meaningful perturbations to the gradients of deep neural networks. Yang et al. [32] validate that the adversarial examples generated by ALERT are valuable in improving the robustness of victim models. Srikant et al. [33] validate that their proposed attack generation method improves adversarial robustness when used to train the SEQ2SEQ model adversarially. Tian et al. [38]

validate the capability of enhancing model robustness after adversarial fine-tuning. Nguyen et al. [64] validate that GraphCodeAttack's fine-tuning, although not requiring the target models' feedback nor a full adversarial attack on the training dataset like ALERT and CARROT, achieves similar performance to ALERT and better performance in comparison with CARROT. Gao et al. [48] propose Enhanced Adversarial Training (EverI), which generally applies to defending against any adversarial attack in principle. Zhou et al. [51] conclude that the masked training method can significantly boost robustness across different models while maintaining fairly good performance on the test dataset. Zhou et al. [37] demonstrate that CARROT can estimate a tight bound of the true robustness of DL models. Liu et al. [16] conclude that their work can guide the development of more robust identification methods. Chen et al. [57] conclude that adversarial training using the adversarial examples generated by their attack method can reduce the attack success rates of source code processing models. In conclusion, using robust adversarial training effectively enhances the robustness of DSCM.

7 Ethical and security implications of robustness attacks targeting deep code models

Robustness attacks on code models can have significant ethical and security implications. While these techniques are intended to evaluate and improve models' robustness, they can be misused for malicious purposes. Ethically, the misuse of robustness attack methods could lead to severe consequences. For instance, if adversarial examples are crafted to target mission-critical systems like autonomous vehicles or medical diagnosis models, they could cause catastrophic failures, putting human lives at risk. Additionally, poisoning attacks on code models used in software development or cybersecurity applications could introduce vulnerabilities or backdoors, compromising the security and integrity of critical systems.

Moreover, the ability to extract sensitive information from code models through privacy violation attacks raises significant privacy concerns. If an attacker can reconstruct training data or model parameters, it could expose confidential code, intellectual property, or personal information, leading to data breaches and potential legal and financial consequences. From a security perspective, the proliferation of robustness attack techniques in the wrong hands could pose severe threats to various industries and infrastructure. Malicious actors could leverage these methods to launch targeted attacks on code models powering critical systems, potentially leading to system failures, data corruption, or unauthorized access.

To mitigate these risks, it is crucial to implement robust security measures and safeguards. This includes enhancing the robustness and resilience of code models through advanced defense techniques, implementing strict access controls and authentication mechanisms, and establishing comprehensive security protocols and incident response plans. Additionally, ethical guidelines and regulatory frameworks should be developed to govern the responsible use of robustness attack methods. These techniques should be strictly utilized for legitimate purposes, such as model evaluation, security testing, and research, under controlled environments and with proper oversight.

By addressing these ethical and security implications proactively, we can harness the potential of robustness attack methods to improve the safety and reliability of code models while mitigating the risks of misuse and ensuring the responsible development and deployment of these technologies.

8 Future Research Directions

In the previous sections, we have comprehensively analyzed recent research on robustness attacks on deep code models. It is evident that many critical problems still need to be addressed. This section provides an outlook on key directions for future research on robustness attacks.

1. **More applications of DSCM:** Robustness attacks, as a method, pose security threats and can also play a role in other directions. Some beneficial applications have emerged, but many potential applications exist in different domains. For example, robustness attack techniques can be used for data augmentation, especially for vulnerability data with high labeling costs, to improve deep code models' robustness and generalization ability.The potential applications for enhancing deep learning models include: (1) How to combine robustness attack methods with adversarial training to generate deep code models with stronger robustness; (2) Explore the balance between robustness metrics and model performance, maximizing robustness while ensuring model performance; (3) Study specialized robustness enhancement solutions for different types of deep code models (such as code translation, code completion, code summarization, etc.).
2. **More realistic attacks:** There is limited research on robustness attacks against black-box models and the real physical world. Such attacks pose a greater threat and are more meaningful to study. Making robustness attack methods more stealthy under black-box model conditions and better resistant to potential interference from the physical world is a significant challenge for the future.
3. **Quantitative risk assessment of DSCM:** Quantifying the risk of robustness attacks faced by deep code models is crucial for developing security strategies. Traditional assessment methods include 1) rule-based assessment, formulating a series of rules to examine whether the model meets expectations; 2) test case-based assessment, constructing test samples under different scenarios to evaluate the model's performance; 3) metric-based assessment, designing relevant metrics to quantify the model's security performance. These methods have advantages and disadvantages, and it is difficult to fully assess all risks of deep code models. Designing an effective model security assessment framework and metrics for robustness attacks is also a major problem that needs to be addressed when studying robustness attacks on deep code models.
4. **Interpretability of DSCM:** Robustness attacks are based solely on experimental results without complete and effective theoretical support. What types of models are more susceptible to robustness attacks, and what types of adversarial examples are more easily learned by models? Discussions and analyses related to interpretability are also a major research direction.

5. **Technical challenges to be urgently solved:** (1) The lack of large-scale, high-quality adversarial code sample sets affects the depth and breadth of robustness attack and defense research; (2) The existing robustness evaluation metrics and methods are not mature enough, making it difficult to comprehensively evaluate the robustness of models; (3) Improving model robustness while ensuring its performance does not significantly decrease is a major challenge in balancing the two; (4) There are large differences in robustness characteristics among different types of code models, and there is a lack of a universal robustness enhancement scheme.
6. **Exploring the potential of interdisciplinary methods:** (1) Introducing game theory into robustness attack and defense research, constructing game models for both attackers and defenders and exploring equilibrium strategies; (2) Drawing on research methods from cognitive psychology and human-computer interaction to analyze the robustness characteristics of code from a human cognitive perspective; (3) Using natural language processing and knowledge graph techniques to enhance code semantic understanding and improve robustness; (4) Exploring the combination of formal verification methods and neural networks to improve the theoretical guarantee of robustness.

9 Conclusion

This paper comprehensively analyzes and classifies existing robustness attack methods for DSCM and summarizes commonly used datasets and models. Currently, robustness attacks on DSCM are still immature, with many researchable directions. It is a rapidly developing field in deep learning, and we hope that this paper can provide a summary reference for future research on robustness attacks.

We hereby call on researchers and practitioners to pay attention to the gaps and future research directions summarized in this article. Although some progress has been made in the research of robustness attacks on deep code models, many problems and challenges still need to be explored. This article analyzes the limitations of existing work, such as the lack of large-scale, high-quality adversarial code sample sets, the need to improve robustness evaluation metrics and methods, and the difficulty in balancing the improvement of robustness with the guarantee of performance. These are key research gaps that need to be solved urgently, and they have important theoretical significance and practical value, requiring researchers to devote more effort to overcome them. In the future, with the continuous filling of research gaps, the robustness of deep code models will surely be comprehensively improved, greatly promoting the application of artificial intelligence technology in software engineering. More reliable, secure, and intelligent deep learning models will inject new impetus into software development, significantly improving development efficiency and quality and bringing huge economic benefits and social value.

Declarations

• Funding This work was supported by Guangxi Key Laboratory of Trusted Software(No.kx202046),2023 Higher Education Scientific Research Planning Project of China Society of Higher Education (No.23PG0408), 2023 Philosophy and Social

Science Research Programs in Jiangsu Province (No.2023SJSZ0993), Nantong Science and Technology Project (No.JC2023070), and the Fifth Research Project on Teaching Reform of Vocational Education in Jiangsu Province (No.ZYB686). This work was also supported by the National Natural Science Foundation of China (No. U23A20280,62172350), the Guangdong Basic and Applied Basic Research Foundation (No.2023A1515012846), the Key Research and Development Program of Guangxi (No.2023AB29051) and Open Project Program of the State Key Laboratory for Novel Software Technology (Nanjing University) (No.KFKT2022B10). This work is sponsored by the Qing Lan Project of Jiangsu Province and sponsored by the Cultivation of Young and Middle-aged Academic Leaders in "Qing Lan Project" of Jiangsu Province.

References

[1] Zakeri-Nasrabadi, M., Parsa, S., Ramezani, M., Roy, C., Ekhtiarzadeh, M.: A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. Journal of Systems and Software, 111796 (2023). Accessed 2024-03-17

[2] Weisz, J.D., Muller, M., Ross, S.I., Martinez, F., Houde, S., Agarwal, M., Talamadupula, K., Richards, J.T.: Better Together? An Evaluation of AI-Supported Code Translation. In: 27th International Conference on Intelligent User Interfaces, pp. 369–391. ACM, Helsinki Finland (2022). https://doi.org/10.1145/3490099.3511157 . https://dl.acm.org/doi/10.1145/3490099.3511157 Accessed 2024-03-17

[3] Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., Jin, H.: A comparative study of deep learning-based vulnerability detection system. IEEE Access **7**, 103184–103197 (2019). Accessed 2024-03-17

[4] Wu, F., Wang, J., Liu, J., Wang, W.: Vulnerability detection with deep learning. In: 2017 3rd IEEE International Conference on Computer and Communications (ICCC), pp. 1298–1302. IEEE, ??? (2017). https://ieeexplore.ieee.org/abstract/document/8322752/ Accessed 2024-03-17

[5] Lin, G., Wen, S., Han, Q.-L., Zhang, J., Xiang, Y.: Software vulnerability detection using deep neural networks: a survey. Proceedings of the IEEE **108**(10), 1825–1848 (2020). Accessed 2024-03-17

[6] Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering **48**(9), 3280–3296 (2021). Accessed 2024-03-17

[7] Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Communications of the ACM **53**(5), 109–116 (2010) https://doi.org/10.1145/1735223.1735249 . Accessed 2024-03-17

[8] Le Goues, C., Pradel, M., Roychoudhury, A., Chandra, S.: Automatic program

repair. IEEE Software **38**(4), 22–27 (2021). Accessed 2024-03-17

[9] Ye, H., Martinez, M., Durieux, T., Monperrus, M.: A comprehensive study of automatic program repair on the QuixBugs benchmark. Journal of Systems and Software **171**, 110825 (2021). Accessed 2024-03-17

[10] Zhang, C., Wang, J., Zhou, Q., Xu, T., Tang, K., Gui, H., Liu, F.: A survey of automatic source code summarization. Symmetry **14**(3), 471 (2022). Accessed 2024-03-17

[11] Zhu, Y., Pan, M.: Automatic Code Summarization: A Systematic Literature Review. arXiv. arXiv:1909.04352 [cs] (2019). http://arxiv.org/abs/1909.04352 Accessed 2024-03-17

[12] LeClair, A., Haque, S., Wu, L., McMillan, C.: Improved Code Summarization via a Graph Neural Network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195. ACM, Seoul Republic of Korea (2020). https://doi.org/10.1145/3387904.3389268 . https://dl.acm.org/doi/10.1145/3387904.3389268 Accessed 2024-03-17

[13] Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.-W.: A Transformer-based Approach for Source Code Summarization. arXiv. arXiv:2005.00653 [cs, stat] (2020). http://arxiv.org/abs/2005.00653 Accessed 2024-03-17

[14] McBurney, P.W., McMillan, C.: Automatic source code summarization of context for java methods. IEEE Transactions on Software Engineering **42**(2), 103–119 (2015). Accessed 2024-03-17

[15] Rabin, M.R.I., Bui, N.D.Q., Wang, K., Yu, Y., Jiang, L., Alipour, M.A.: On the Generalizability of Neural Program Models with respect to Semantic-Preserving Program Transformations. Information and Software Technology **135**, 106552 (2021) https://doi.org/10.1016/j.infsof.2021.106552 . Accessed 2023-08-15

[16] Liu, Q., Ji, S., Liu, C., Wu, C.: A Practical Black-Box Attack on Source Code Authorship Identification Classifiers. IEEE Transactions on Information Forensics and Security **16**, 3620–3633 (2021) https://doi.org/10.1109/TIFS.2021.3080507

[17] Hammad, M., Babur, o., Basit, H.A.: Augmenting Machine Learning with Information Retrieval to Recommend Real Cloned Code Methods for Code Completion. arXiv. arXiv:2010.00964 [cs] (2020). http://arxiv.org/abs/2010.00964 Accessed 2024-03-17

[18] Lee, C., Gottschlich, J., Roth, D.: Toward Code Generation: A Survey and Lessons from Semantic Parsing. arXiv. arXiv:2105.03317 [cs] (2021). https://doi.org/10.48550/arXiv.2105.03317 . http://arxiv.org/abs/2105.03317 Accessed 2024-03-17

[19] Ahmed, A., Azab, S., Abdelhamid, Y.: Source-Code Generation Using Deep

Learning: A Survey. In: Progress in Artificial Intelligence: 22nd EPIA Conference on Artificial Intelligence, EPIA 2023, Faial Island, Azores, September 5–8, 2023, Proceedings, Part II, pp. 467–482. Springer, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-49011-8_37

[20] Qu, Y., Wong, W.E., Li, D.: Empirical research for self-admitted technical debt detection in blockchain software projects. International Journal of Performability Engineering **18**(3), 149 (2022). Accessed 2024-03-27

[21] Yang, Z., Sun, Z., Yue, T.Z., Devanbu, P., Lo, D.: Robustness, Security, Privacy, Explainability, Efficiency, and Usability of Large Language Models for Code. arXiv. arXiv:2403.07506 [cs] (2024). https://doi.org/10.48550/arXiv.2403.07506 . http://arxiv.org/abs/2403.07506 Accessed 2024-03-16

[22] Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C.C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., Synnaeve, G.: Code Llama: Open Foundation Models for Code. arXiv. arXiv:2308.12950 [cs] (2024). https://doi.org/10.48550/arXiv.2308.12950 . http://arxiv.org/abs/2308.12950 Accessed 2024-03-16

[23] Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T.Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L.K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S.S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C.J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C.M., Hughes, S., Wolf, T., Guha, A., Werra, L., Vries, H.: StarCoder: may the source be with you! arXiv. arXiv:2305.06161 [cs] (2023). https://doi.org/10.48550/arXiv.2305.06161 . http://arxiv.org/abs/2305.06161 Accessed 2024-03-16

[24] Microsoft: GitHub Copilot · Your AI pair programmer (2023). https://github.com/features/copilot Accessed 2024-03-16

[25] Amazon: AI Code Generator - Amazon CodeWhisperer - AWS. https://aws.amazon.com/codewhisperer/ Accessed 2024-03-16

[26] Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Wang, Z., Shen, L., Wang, A., Li, Y., Su, T., Yang, Z., Tang, J.: CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. arXiv (2023). http://arxiv.org/abs/2303.17568 Accessed 2023-06-28

[27] TabNine: Tabnine is an AI assistant that speeds up delivery and keeps your code safe (2023). https://www.tabnine.com/ Accessed 2024-03-16

[28] aiXcoder: aiXcoder. https://www.aixcoder.com/#/ Accessed 2024-03-16

[29] Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y., Jin, Z.: Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. Proceedings of the AAAI Conference on Artificial Intelligence **34**(01), 1169–1176 (2020) https://doi.org/10.1609/aaai.v34i01.5469 . tex.copyright: Copyright (c) 2020 Association for the Advancement of Artificial Intelligence. Accessed 2023-04-21

[30] Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019) https://doi.org/10.1145/3290353 . Accessed 2023-12-17

[31] Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating Sequences from Structured Representations of Code. arXiv (2019). https://doi.org/10.48550/arXiv.1808.01400 . http://arxiv.org/abs/1808.01400 Accessed 2023-08-13

[32] Yang, Z., Shi, J., He, J., Lo, D.: Natural attack for pre-trained models of code. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, pp. 1482–1493. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510146 . https://dl.acm.org/doi/10.1145/3510003.3510146 Accessed 2023-04-22

[33] Srikant, S., Liu, S., Mitrovska, T., Chang, S., Fan, Q., Zhang, G., O'Reilly, U.-M.: Generating Adversarial Computer Programs using Optimized Obfuscations. arXiv. arXiv:2103.11882 [cs] (2021). http://arxiv.org/abs/2103.11882 Accessed 2024-03-29

[34] Yefet, N., Alon, U., Yahav, E.: Adversarial Examples for Models of Code. arXiv (2020). https://doi.org/10.48550/arXiv.1910.07517 . http://arxiv.org/abs/1910.07517 Accessed 2023-05-07

[35] Pour, M.V., Li, Z., Ma, L., Hemmati, H.: A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding. arXiv. arXiv:2101.07910 [cs] (2021). https://doi.org/10.48550/arXiv.2101.07910 . http://arxiv.org/abs/2101.07910 Accessed 2024-04-03

[36] Jha, A., Reddy, C.K.: CodeAttack: Code-Based Adversarial Attacks for Pre-trained Programming Language Models. arXiv. arXiv:2206.00052 [cs] (2023). https://doi.org/10.48550/arXiv.2206.00052 . http://arxiv.org/abs/2206.00052 Accessed 2024-04-04

[37] Zhang, H., Fu, Z., Li, G., Ma, L., Zhao, Z., Yang, H., Sun, Y., Liu, Y., Jin, Z.: Towards Robustness of Deep Program Processing Models—Detection, Estimation, and Enhancement. ACM Transactions on Software Engineering and

Methodology (TOSEM) **31**, 1–40 (2022)

[38] Tian, Z., Chen, J., Jin, Z.: Code Difference Guided Adversarial Example Generation for Deep Code Models. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 850–862. IEEE, Luxembourg, Luxembourg (2023). https://doi.org/10.1109/ASE56229.2023.00149 . https://ieeexplore.ieee.org/document/10298520/ Accessed 2024-03-29

[39] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating Large Language Models Trained on Code. arXiv. arXiv:2107.03374 [cs] (2021). https://doi.org/10.48550/arXiv.2107.03374 . http://arxiv.org/abs/2107.03374 Accessed 2024-03-18

[40] Wang, Y., Wang, W., Joty, S., Hoi, S.C.H.: CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv (2021). https://doi.org/10.48550/arXiv.2109.00859 . http://arxiv.org/abs/2109.00859 Accessed 2023-09-26

[41] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A.M., Pillai, T.S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., Fiedel, N.: PaLM: Scaling Language Modeling with Pathways. arXiv. arXiv:2204.02311 [cs] (2022). https://doi.org/10.48550/arXiv.2204.02311 . http://arxiv.org/abs/2204.02311 Accessed 2024-03-18

[42] He, J., Vechev, M.: Large Language Models for Code: Security Hardening and Adversarial Testing. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 1865–1879 (2023). https://doi.org/10.1145/3576915.3623175 . arXiv:2302.05319 [cs]. http://arxiv.org/abs/2302.05319 Accessed 2024-03-16

[43] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. arXiv. arXiv:2108.09293 [cs] (2021). https://doi.org/10.48550/arXiv.2108.09293 . http://arxiv.org/abs/2108.09293 Accessed 2024-02-12

[44] Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., Lewis, M.: InCoder: A Generative Model for Code Infilling and Synthesis. arXiv. arXiv:2204.05999 [cs] (2023). https://doi.org/10.48550/arXiv.2204.05999 . http://arxiv.org/abs/2204.05999 Accessed 2024-02-12

[45] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv. arXiv:2203.13474 [cs] (2023). https://doi.org/10.48550/arXiv.2203.13474 . http://arxiv.org/abs/2203.13474 Accessed 2024-02-12

[46] Biggio, B., Fumera, G., Roli, F.: Security evaluation of pattern classifiers under attack. IEEE transactions on knowledge and data engineering **26**(4), 984–996 (2013). Accessed 2024-01-01

[47] Bielik, P., Vechev, M.: Adversarial Robustness for Code. arXiv. arXiv:2002.04694 [cs, stat] (2020). https://doi.org/10.48550/arXiv.2002.04694 . http://arxiv.org/abs/2002.04694 Accessed 2024-03-30

[48] Gao, F., Wang, Y., Wang, K.: Discrete Adversarial Attack to Models of Code. Proceedings of the ACM on Programming Languages **7**(PLDI), 113–172113195 (2023) https://doi.org/10.1145/3591227 . Accessed 2023-06-12

[49] Yang, G., Zhou, Y., Yang, W., Yue, T., Chen, X., Chen, T.: How Important Are Good Method Names in Neural Code Generation? A Model Robustness Perspective. ACM Transactions on Software Engineering and Methodology **33**(3), 60–16035 (2024) https://doi.org/10.1145/3630010 . Accessed 2024-04-04

[50] Quiring, E., Maier, A., Rieck, K.: Misleading Authorship Attribution of Source Code using Adversarial Learning. arXiv. arXiv:1905.12386 [cs, stat] (2019). https://doi.org/10.48550/arXiv.1905.12386 . http://arxiv.org/abs/1905.12386 Accessed 2024-03-30

[51] Zhou, Y., Zhang, X., Shen, J., Han, T., Chen, T., Gall, H.: Adversarial Robustness of Deep Code Comment Generation. ACM Transactions on Software Engineering and Methodology **31**(4), 1–30 (2022) https://doi.org/10.1145/3501256 . Accessed 2023-08-20

[52] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. 2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings (2013) https://doi.org/10.48550/arxiv.1312.6199

[53] Carlini, N., Wagner, D.: Audio adversarial examples: Targeted attacks on speech-to-text. In: 2018 IEEE Security and Privacy Workshops (SPW), pp. 1–7. IEEE, ??? (2018). https://ieeexplore.ieee.org/abstract/document/8424625/ Accessed 2023-12-17

[54] Zhang, J., Ma, W., Hu, Q., Liu, S., Xie, X., Traon, Y.L., Liu, Y.: A Black-Box Attack on Code Models via Representation Nearest Neighbor Search. arXiv. arXiv:2305.05896 [cs] (2023). https://doi.org/10.48550/arXiv.2305.05896 . http://arxiv.org/abs/2305.05896 Accessed 2024-04-03

[55] Na, C., Choi, Y., Lee, J.-H.: DIP: Dead code Insertion based Black-box Attack for Programming Language Model. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 7777–7791. Association for Computational Linguistics, Toronto, Canada (2023). https://doi.org/10.18653/v1/2023.acl-long.430 . https://aclanthology.org/2023.acl-long.430

[56] Choi, Y., Kim, H., Lee, J.-H.: TABS: Efficient Textual Adversarial Attack for Pre-trained NL Code Model Using Semantic Beam Search. In: Conference on Empirical Methods in Natural Language Processing (2022). https://api.semanticscholar.org/CorpusID:256461306

[57] Chen, P., Li, Z., Wen, Y., Liu, L.: Generating Adversarial Source Programs Using Important Tokens-based Structural Transformations. In: 2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 173–182. IEEE, ??? (2022). https://ieeexplore.ieee.org/abstract/document/9763729/ Accessed 2023-12-21

[58] Tian, J., Wang, C., Li, Z., Wen, Y.: Generating Adversarial Examples of Source Code Classification Models via Q-Learning-Based Markov Decision Process. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 807–818. IEEE, ??? (2021). https://ieeexplore.ieee.org/abstract/document/9724884/ Accessed 2024-01-01

[59] Ramakrishnan, G., Henkel, J., Wang, Z., Albarghouthi, A., Jha, S., Reps, T.: Semantic Robustness of Models of Source Code. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 526–537 (2022). https://doi.org/10.1109/SANER53432.2022.00070 . http://arxiv.org/abs/2002.03043 Accessed 2023-08-14

[60] Zhang, W., Guo, S., Zhang, H., Sui, Y., Xue, Y., Xu, Y.: Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations. IEEE Transactions on Software Engineering **49**(5), 3052–3070 (2023) https://doi.org/10.1109/TSE.2023.3240118

[61] Yang, G., Zhou, Y., Zhang, X., Chen, X., Han, T., Chen, T.: Assessing and improving syntactic adversarial robustness of pre-trained models for

code translation. arXiv: 2310.18587 [id='cs.SE' full_name='Software Engineering' is_active=True alt_name=None in_archive='cs' is_general=False description='Covers design tools, software metrics, testing and debugging, programming environments, etc. Roughly includes material in all of ACM Subject Classes D.2, except that D.2.4 (program verification) should probably have Logics in Computer Science as the primary subject area.'] (2023)

[62] Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: Learning to Write Programs. arXiv. arXiv:1611.01989 [cs] (2017). https://doi.org/10.48550/arXiv.1611.01989 . http://arxiv.org/abs/1611.01989 Accessed 2024-03-30

[63] Ebrahimi, J., Rao, A., Lowd, D., Dou, D.: HotFlip: White-Box Adversarial Examples for Text Classification. arXiv. arXiv:1712.06751 [cs] (2018). https://doi.org/10.48550/arXiv.1712.06751 . http://arxiv.org/abs/1712.06751 Accessed 2024-03-30

[64] Nguyen, T.-D., Zhou, Y., Le, X.B.D., Patanamon, Thongtanunam, Lo, D.: Adversarial Attacks on Code Models with Discriminative Graph Patterns. arXiv. arXiv:2308.11161 [cs] (2023). https://doi.org/10.48550/arXiv.2308.11161 . http://arxiv.org/abs/2308.11161 Accessed 2024-01-06

[65] Wang, K., Christodorescu, M.: COSET: A Benchmark for Evaluating Neural Program Embeddings. arXiv. arXiv:1905.11445 [cs, stat] (2019). https://doi.org/10.48550/arXiv.1905.11445 . http://arxiv.org/abs/1905.11445 Accessed 2024-03-30

[66] Pierazzi, F., Pendlebury, F., Cortellazzi, J., Cavallaro, L.: Intriguing Properties of Adversarial ML Attacks in the Problem Space. arXiv. arXiv:1911.02142 [cs] (2020). https://doi.org/10.48550/arXiv.1911.02142 . http://arxiv.org/abs/1911.02142 Accessed 2024-03-30

[67] Liu, H., Sun, C., Su, Z., Jiang, Y., Gu, M., Sun, J.: Stochastic Optimization of Program Obfuscation. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 221–231 (2017). https://doi.org/10.1109/ICSE.2017.28 . ISSN: 1558-1225. https://ieeexplore.ieee.org/document/7985664 Accessed 2024-03-30

[68] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M.: CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv. arXiv:1909.09436 [cs, stat] (2020). https://doi.org/10.48550/arXiv.1909.09436 . http://arxiv.org/abs/1909.09436 Accessed 2024-04-05

[69] Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 30 (2016). Issue: 1. https://ojs.aaai.org/index.php/AAAI/article/view/10139 Accessed 2024-04-05

[70] Alsulami, B., Dauber, E., Harang, R., Mancoridis, S., Greenstadt, R.: Source Code Authorship Attribution Using Long Short-Term Memory Based Networks. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) Computer Security – ESORICS 2017, pp. 65–82. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66402-6_6

[71] Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M.: Towards a Big Data Curated Benchmark of Inter-project Code Clones. In: 2014 IEEE International Conference on Software Maintenance And Evolution, pp. 476–480 (2014). https://doi.org/10.1109/ICSME.2014.77 . ISSN: 1063-6773. https://ieeexplore.ieee.org/abstract/document/6976121 Accessed 2024-04-05

[72] Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Mapping Language to Code in Programmatic Context. arXiv. arXiv:1808.09588 [cs] (2018). http://arxiv.org/abs/1808.09588 Accessed 2024-04-05

[73] Yin, P., Deng, B., Chen, E., Vasilescu, B., Neubig, G.: Learning to mine aligned code and natural language pairs from stack overflow. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 476–486. ACM, Gothenburg Sweden (2018). https://doi.org/10.1145/3196398.3196408 . https://dl.acm.org/doi/10.1145/3196398.3196408 Accessed 2024-04-05

[74] Nikitopoulos, G., Dritsa, K., Louridas, P., Mitropoulos, D.: CrossVul: a cross-language vulnerability dataset with commit data. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1565–1569. ACM, Athens Greece (2021). https://doi.org/10.1145/3468264.3473122 . https://dl.acm.org/doi/10.1145/3468264.3473122 Accessed 2024-04-05

[75] Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 508–512. ACM, Seoul Republic of Korea (2020). https://doi.org/10.1145/3379597.3387501 . https://dl.acm.org/doi/10.1145/3379597.3387501 Accessed 2023-12-18

[76] Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., Grunske, L.: VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python. Information and Software Technology **144**, 106809 (2022) https://doi.org/10.1016/j.infsof.2021.106809 . Accessed 2024-04-05

[77] Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D.: An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. arXiv. arXiv:1812.08693 [cs] (2019). http://arxiv.org/abs/1812.08693 Accessed 2024-03-29

[78] Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z.: Summarizing source code with transferred api knowledge (2018). Accessed 2024-04-05

[79] Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S.: Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 397–407. ACM, Montpellier France (2018). https://doi.org/10.1145/3238147.3238206 . https://dl.acm.org/doi/10.1145/3238147.3238206 Accessed 2024-04-05

[80] Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to Represent Programs with Graphs. (2018). https://openreview.net/forum?id=BJOFETxR- Accessed 2024-04-05

[81] Raychev, V., Bielik, P., Vechev, M.: Probabilistic model for code with decision trees. ACM SIGPLAN Notices **51**(10), 731–747 (2016) https://doi.org/10.1145/3022671.2984041 . Accessed 2024-04-05

[82] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. arXiv (2019). https://doi.org/10.48550/arXiv.1909.03496 . http://arxiv.org/abs/1909.03496 Accessed 2023-04-22

[83] Dong, Z., Hu, Q., Guo, Y., Zhang, Z., Cordy, M., Papadakis, M., Traon, Y.L., Zhao, J.: Boosting Source Code Learning with Data Augmentation: An Empirical Study. arXiv. arXiv:2303.06808 [cs] (2023). https://doi.org/10.48550/arXiv.2303.06808 . http://arxiv.org/abs/2303.06808 Accessed 2024-04-04

[84] Li, Y., Wu, H., Zhao, H.: Semantic-Preserving Adversarial Code Comprehension. arXiv (2022). https://doi.org/10.48550/arXiv.2209.05130 . http://arxiv.org/abs/2209.05130 Accessed 2023-08-19

[85] Li, Z., Guenevere, Chen, Chen, C., Zou, Y., Xu, S.: RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1906–1918 (2022). https://doi.org/10.1145/3510003.3510181 . http://arxiv.org/abs/2202.06043 Accessed 2023-05-29