Mutual Learning-Based Framework for Enhancing Robustness of Code Models via Adversarial Training

Yangsen Wang*
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
wangyangsen@stu.pku.edu.cn

Yizhou Chen
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
yizhouchen@stu.pku.edu.cn

Yifan Zhao
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhaoyifan@stu.pku.edu.cn

Zhihao Gong
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhihaogong@stu.pku.edu.cn

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

Dan Hao†
School of Electronic and Computer
Engineering, Peking University
Shenzhen, China
haodan@pku.edu.cn

ABSTRACT

Deep code models (DCMs) have achieved impressive accomplishments and have been widely applied to various code-related tasks. However, existing studies show that some DCMs have poor robustness, and even small noise in the input data can lead to erroneous outputs. This phenomenon can seriously hinder the application of these DCMs in real-world scenarios. To address this limitation, we propose MARVEL, a mutual learning-based framework for enhancing the robustness of DCMs via adversarial training. Specifically, MARVEL initializes two identical DCMs, one of which receives Gaussian-distorted data and performs adversarial training, and the other receives the clean data. Then these two DCMs work together to not only fit the true labels but also fit each other's internal parameters. Our intuition is that the DCM can enhance robustness by training noisy data, while the DCM achieves accurate prediction performance by learn the clean data. Their mutual learning enables the DCM to balance both robustness and predictive performance.

We selected three popular DCMs, five open-source datasets, and three state-of-the-art attack methods to evaluate the performance of MARVEL on 45 (3×5×3) downstream tasks composed of their combinations. Additionally, we set two of the state-of-the-art robustness enhancement techniques as baselines. The experimental results show that MARVEL significantly enhances the robustness of DCMs across all 45 tasks. In 43 out of 45 tasks, MARVEL outperforms the two baselines with an average improvement of 15.33% and 31.88%, respectively. At the same time, MARVEL can maintain the inherent accuracy with an error margin within +-2.43% compared to the original DCMs.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

Code Model, Deep Mutual Learning, Model Robustness, Adversarial Training

ACM Reference Format:
Yangsen Wang, Yizhou Chen, Yifan Zhao, Zhihao Gong, Junjie Chen, and Dan Hao. 2024. Mutual Learning-Based Framework for Enhancing Robustness of Code Models via Adversarial Training. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695519

1 INTRODUCTION

Over the recent years, Deep Learning (DL) has achieved impressive results in various tasks of software engineering, including code clone detection [19, 34, 55], functionality classification [65], vulnerability detection [9, 11, 35, 67, 69] and others [13, 14, 58]. Many deep learning models such as CodeBERT [20], GraphCodeBERT [27], UniXCoder [26], and others have been developed based on large-scale code snippets, and according to existing work, we refer these models as Deep Code Models (DCMs) [49, 50]. However, the superior performance demonstrated by DCMs during the testing stage may not necessarily translate to the equivalent level of performance when confronted with real-world inputs encountered in actual development environments [17, 61]. In practice, program coding and design specifications are not universal, but depend on the requirements and conventions of particular projects. Programming languages only need to conform to their syntax and logic rules in order to function. As a result, some code may introduce potential noise for DCMs, including inconsistent identifier names [56], redundant dead code [44], and disorganized code structures

*The author is also affiliated with College of Intelligence and Computing, Tianjin University.
†Corresponding author.
HCST: High Confidence Software Technologies.
SCS: School of Computer Science.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695519

[49], etc. These attributes can interfere with the output results of DCM, as they are infrequently encountered in curated training data by researchers.

Several researchers have identified such weaknesses in DCMs and proposed diverse adversarial attack techniques [49, 56, 57, 62, 64], aiming to emulate real-world noise through strategies like identifier replacement, dead code injection, and equivalent structure transformation within the code. Subsequently, these noisy data are inputted into the DCMs, leading to erroneous outputs. For example, the state-of-the-art attack technique [49] achieves an average attack success rate (outputting erroneous results) of 73.04% when targeting CodeBERT [20]. That is, when both the original data and the noise-disturbed data are fed into CodeBERT, it is discovered that approximately 73.04% of the predictions with noise-disturbed data are inconsistent with the original data. This further demonstrates that existing DCMs have the limitation of not being robust enough in the face of real-world development scenarios. As such, it is crucial to ensure the robustness of these DCMs.

In view of the above, many researchers work on improving robustness of DCMs and their work can be divided into two categories: data-level approaches [16, 49, 56, 59, 62, 63] and model-level approaches [7, 21, 28, 31, 36–38]. The data-level approaches typically begin by generating adversarial examples and subsequently employ them to iteratively fine-tune the model, thereby enhancing its robustness [49, 56, 62]. **However, these generated adversarial examples often do not fully cover the features of real-world noise.** For instance, altering variable names represents a straightforward means of perturbing the model, nevertheless, variable names may have diverse naming conventions, while real noise may be more complex and varied, far beyond what model could learn from adversarial examples. In contrast, the model-level approach does not introduce additional training data, some existing model-level approaches employ adversarial training [7, 28, 31, 36], while some employ advanced network framework to enhance the robustness of DCMs [21, 37, 38]. **However, their effectiveness against multiple adversarial attacks remains uncertain due to the limited reporting of test results for only one or a few types of adversarial attacks.** Another point to note is that the noisy data introduced in adversarial training will inevitably cause biases in the expected results. These biases may lead to a decrease in the accuracy of the DCMs. **The trade-off between accuracy and robustness must be meticulously considered when conducting adversarial training.**

To address the aforementioned limitations, we propose a novel **M**utual le**AR**ning ad**VE**rsaria**L** method, named MARVEL. In brief, MARVEL consists of three important modules, including the Hidden space Data Augmentation (HidDA), the Mutual learning Adversarial Training (MutAT), and the Mutual learning Feature Fusion (MutFF). The HidDA simulates various noise disturbances that the input data may face by directly injecting a certain degree of Gaussian noise for DCM in the hidden space, and incorporates these Gaussian noises into the model training to improve the robustness of the DCM. In addition, MutAT inputs two sets of data, i.e., clean and noisy data, into two same DCM respectively. The trade-off between accuracy and robustness is maintained by learning the data distribution of each other. The MutFF ultimately integrates the outputs of the two DCMs and fits the true labels using a multi-layer perceptron. It is important to emphasise that our MARVEL is not only applicable to specific DCM, but has the potential to be applied to a wide range of different DCMs to enhance their robustness. This feature makes MARVEL a generic solution that can support multiple DCMs and their downstream tasks.

In order to demonstrate the effectiveness of MARVEL, we conducted an extensive evaluation on three widely used DCMs (i.e., CodeBERT [20], GraphCodeBERT [27] and UniXCoder [26]) and five datasets covering three popular languages (i.e., Python, Java, and C). For each combination of datasets and DCMs, we selected three state-of-the-art adversarial attack methods (i.e., ALERT [56], MHM[63] and CODA [49]) to attack the MARVEL-enhanced models. And two state-of-the-art robustness enhancement techniques are chosen as the baselines. The degree of reduction in the attack success rates (ASR) of these attack methods is used as a quantitative indicator of robustness. The experimental results show that MARVEL can improve the robustness of DCMs in all 45 tasks (combinations of three DCMs, five datasets, and three adversarial attack methods). In 43 of 45 tasks, MARVEL surpass the baseline methods. Compared to the baseline methods CREAM[21] and SPACE[36], our MARVEL improves robustness by an average of 15.19% and 31.80%, respectively. Furthermore, the strengthened DCMs by MARVEL maintain a high level of accuracy, with an error margin within +-2.43% compared to the original DCMs.

In summary, the main contributions of this paper can be summarized as follows:

- We propose MARVEL, a robustness enhancement technique for DCMs. It can be applied to multiple DCMs and defends against various adversarial attacks.
- We perform an extensive evaluation in terms of the effectiveness of MARVEL on 45 tasks. The results show that the MARVEL-enhanced DCMs successfully defend against more adversarial attacks across all tasks compared to the original DCMs. MARVEL outperforms state-of-the-art robustness enhancement technique on up to 43 tasks, the best of which can improve robustness by up to 43.42%. To the best of our knowledge, we are the first to conduct extensive experiments using latest attack techniques to evaluate DCMs' robustness.
- We open-sourced all our experimental data and code at https://github.com/VMnK-Run/MARVEL, enabling future research and replication of our findings.

2 BACKGROUND AND MOTIVATION

In this section, we provide background about DCMs and deep mutual learning, and present the problem definition and the motivation of our approach.

2.1 Deep Code Models

Deep Neural Networks (DNNs) have been widely used to process source code and have achieved great performance in various downstream tasks [4, 5, 30, 55]. Recently, researchers have proposed some transformer-based pre-trained DCMs [20, 26, 27, 40, 53, 54, 68], through the process of pre-training on large-scale unlabelled code corpora and fine-tuning on specific downstream task datasets, these pre-trained DCMs have achieved state-of-the-art performance on

tasks such as authorship attribution, vulnerability detection, code completion [10].

Feng et al. proposed CodeBERT [20], which is the first bimodal pre-trained model for natural language (NL) and programming language (PL) and can handle multiple programming languages. After that, Guo et al. proposed GraphCodeBERT [27] that uses the same architecture as CodeBERT, but it considers the inherent structure of code and leverages semantic-level information of code, i.e., data flow graph (DFG), for pre-training. Both CodeBERT and GraphCodeBERT are encoder-only architectures. Guo et al. proposed UniXCoder [26] which is a unified cross-modal pre-trained model that can simultaneously support encoder-only, decoder-only, and encoder-decoder modes.

There are many other pre-trained DCMs, including PLBART [3], CodeT5 [54], CodeGPT [40] and so on. In this paper, we mainly focus on the three aforementioned pre-trained DCMs (i.e., CodeBERT [20], GraphCodeBERT [27], UniXCoder [26]), as they have already demonstrated good performance, and investigate their robustness in source code classification tasks such as authorship attribution, defect prediction and functionality classification, following the existing works [28, 49, 50].

2.2 Deep Mutual Learning

To address the need for models with fewer parameters and improved efficiency, researchers have proposed model distillation [29] that transfers knowledge from a larger teacher network to a small student network. Different from this way, Zhang et al. proposed *Deep mutual learning (DML)* [66], which employs multiple models trained simultaneously for the same goal. During the training process, each model not only fits the true label distribution but also leverages the learning experience of other models to further enhance its generalization capability. In other words, each model has two loss functions during the learning process, one is the traditional supervised loss function, which utilizes cross-entropy loss to measure the discrepancy between the model's predicted target class and the true label, the other is the inter-model interactive loss function, which employs Kullback-Leibler (KL) divergence to quantify the divergence between the predicted probability distributions of two networks. The experimental results demonstrate that DML can effectively enhance the performance of individual models.

However, to the best of our knowledge, no prior work has investigated the application of DML in the domain of DCMs. Later we will show that combining the simple DML approach with adversarial training can significantly enhance the robustness of DCMs.

2.3 Problem Definition

Considering source code classification tasks, given a code-label pair $(x, y) \in \mathcal{D}$ and a code model $\mathcal{M}$, where $\mathcal{D}$ represents the test set, x represents a code snippet, and y represents the corresponding correct label, $\mathcal{M}$ can give a probability vector for each input x, where each element represents the probability that $\mathcal{M}$ believes x belongs to the corresponding label. The label with the maximum probability value is the output of $\mathcal{M}$, donoted as $\mathcal{M}$. If $\mathcal{M}(x) = y$, i.e., the code model can correctly classify the given input.

Different from existing work to improve the robustness of DCMs [21, 31, 37, 50], our primary goal is to enhance the ability of DCMs

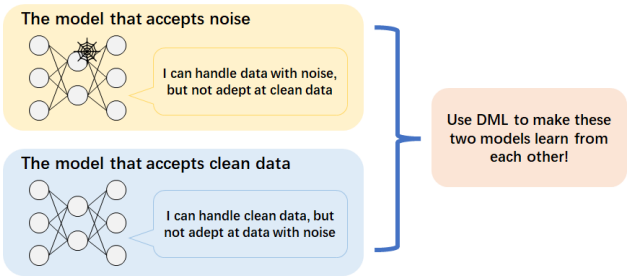

Figure 1: Our motivation for using deep mutual learning.

to defend against state-of-the art adversarial attacks, because these adversarial attacks aim to simulate noise conditions that are as close as possible to real-world development scenarios, which means the model's ability to defend against these adversarial attacks is a more appropriate way to reflect its true robustness. Adversarial attack on code aims to find $x' \in \sigma(x)$ that $\mathcal{M}(x') \neq y$. Here, σ is a function that can modify x in a way that preserves its semantic information and satisfies the grammar constraints with the Frobenius norm constrained to be less than ϵ, and x' is the adversarial example. Only inputs that $\mathcal{M}$ can correctly classify are considered following existing works [49, 56]. Our goal is to make the model able to correctly classify not only the original data, but also the adversarial examples, i.e., successfully defend against adversarial attacks. Formally speaking, we aim to enhance $\mathcal{M}$ to obtain $\mathcal{M}'$, for those samples where $\mathcal{M}(x) = y$, we hope to maintain $\mathcal{M}'(x) = y$ as much as possible. Meanwhile, for the samples where $\mathcal{M}(x) = y \wedge \mathcal{M}(x') \neq y$, we hope to be able to have $\mathcal{M}'(x) = \mathcal{M}'(x') = y$. Since we focus more on the model's robustness to input perturbations, the samples where $\mathcal{M}(x) \neq y$ are not the focus of our consideration. The key focus of this paper is how to obtain a more robust mode $\mathcal{M}'$ and enhance its capability to withstand adversarial attacks.

2.4 Motivation

Existing works [56, 62] have shown that DCMs are highly sensitive to the identifier names in code snippets. As a result, state-of-the-art attack methods targeting DCMs typically involve renaming the identifiers in code snippets to generate adversarial examples that can cause the model to make errors. Existing works also incorporates these adversarial examples back into the training data to fine-tune the DCMs, with the aim of enhancing the robustness of the models. However, from the DNN perspective, code inputs are converted into latent space representations, introducing perturbations to the code inputs corresponds to injecting noise into the model's hidden space. This forces the model to learn stronger decision boundaries to fit the data, thereby improving its robustness. Motivated by this, one basic idea is to directly add noise to the hidden space. Additionally, to prevent the model from overfitting to excessive noise, we propose a gradient-driven hidden space data augmentation module, which adaptively injects Gaussian noise into the model's word embedding layer based on the gradients of the internal parameters, in order to obtain a more robust model.

However, excessively obscuring the embedding representation of the identifiers will affect the model's performance when facing clean data. Therefore, we hope the model can not only withstand

adversarial attacks, but also correctly process clean data. The ideal scenario is to use the model with added noise on the identifiers when facing adversarial attacks, and use the original model when facing clean data. In other words, we want to integrate these two capabilities, which is why we want to introduce DML. As show in Figure 1, we leverage DML to enable the model that can handle noisy data and the model that handle clean data learn from each other, the goal is to achieve the effect of being able to both resist adversarial attacks and correctly fit clean data.

3 APPROACH

3.1 Overview

In this section, we propose MARVEL to against various adversarial attacks. MARVEL improves robustness by working together with three modules included the Hidden Space Data Augmentation (HidDA, Section 3.2), the Mutual Learning Adversarial Training (MutAT, Section 3.3), and the Mutual Learning Feature Fusion (MutFF, Section 3.4). Figure 2 illustrates their functionality and the overall workflow is as follows:

1. **HidDA:** performs data augmentation by adding noise to the high-dimensional embedding space of the code. Then, these noisy data and original data will be fed into the MutAT.
2. **MutAT:** introduces two models with the same structure - one receives original data, and the other receives noisy data for adversarial training, and the two models learn the internal parameters of each other.
3. **MutFF:** integrates the outputs of two models in the MutAT to perform downstream tasks.

In the following, we will elaborate on these modules in detail.

3.2 Hidden Space Data Augmentation

The core idea of adversarial training is to improve the robustness of a model by training it with adversarial examples. These adversarial examples can be small perturbations to the input data or specific injections into the source code. Many researchers introduce [49, 56, 62, 63] inconsistent identifier names, redundant dead code, or other common errors in the original code as adversarial examples. The aforementioned examples, however, often fail to comprehensively simulate real-world errors and merely result in limited enhancements in robustness.

Our intuition is that all adversarial example aims to inject additional perturbations into the model's latent space, which can be mathematically regarded as minor adjustments to both the model parameters and input vectors, therefore, we can simulate complex real-world noise by directly adding noise to the embedding vectors. Based on this idea, we design a HidDA, which adaptively generates Gaussian noise based on the weights of internal parameters within the DCM and introduces it into the word embedding layer. Subsequently, we feed the noisy data into the DCMs for training to enhance their robustness.

Specifically, given a code snippet, we tokenize code snippets into multiple sub-word sequences $Q = \{w_i\}_{i=1}^{n}$ using Byte Pair Encoding Tokenizer (BPE) [47], subsequently, this sequence will be encoded into a high-dimensional embedding space $E = \{e_i\}_{i=1}^{n} \in \mathcal{R}^{n \times d}$, where d is the dimension of embedding space. In this way, we can associate each sub-word with an embedding vector one by one.

The first step MARVEL needs to take is to generate initial noise on the embeddings to perform hidden space data augmentation. To ensure the semantic of the code remains unchanged, MARVEL only adds noise to the embedding vectors corresponding to all identifiers. We believe that this can better simulate the process of generating adversarial examples by continuously modifying identifiers.

MARVEL uses attention information to initialize the noise for the word embeddings corresponding to each identifier. Most existing state-of-the-art DCMs utilize attention mechanisms, which often consist of multiple attention layers. In each attention layer, attention weights are calculated for each token, which can reflect the importance of each token to the model's output. For the convenience of presentation, an identifier list $\mathcal{E} = \{e_1, e_2, ..., e_m\}$ can be obtained for each code snippet Q, and the list of attention layers contained in a code model $\mathcal{M}$ can be denoted as $\mathcal{H} = \{h_1, h_2, ..., h_n\}$, the attention weights assigned to the identifier e_i by the attention layer h_j are represented as w_{ij}. Considering that an identifier may appear multiple times in a code snippet, we take the average of its attention weights at different positions as w_{ij}.

MARVEL uses attention in a way that assigns larger noise to identifiers with higher attention weights. For each identifier e_i, calculate the attention score $s_i = \frac{\sum_{j=1}^{n} w_{ij}}{n}$, where n is the number of attention layers. We use this to assign a new weight $r_i = \frac{m \cdot s_i}{\sum_{i=1}^{m} s_i} + 1$ to each identifier e_i, adding 1 is intended to amplify the initial noise. We first generate random Gaussian noise $z' \in \mathcal{R}^{1 \times d}$, and initialize the noise $\delta_i = z' \cdot r_i$ for each identifier e_i based on s_i. The generated noise will be added to the code embedding representations, in order to achieve hidden space data augmentation. It's worth noting that MARVEL only adds noise to identifiers and ensure that the noise for the same identifier at different positions is the same.

3.3 Mutual Learning Adversarial Training

Introducing noise for training may cause the output of the DCMs to deviate from expectations and lead to a decrease in accuracy. Therefore, we design MutAT to improve robustness while maintaining accuracy.

The workflow of MutAT is shown in the top of Figure 2. We set two identical models, where one model receives noisy data and performs adversarial training, while the other receives the original clean data. And, the two models learn the model parameters of each other during training. The detailed process is illustrated by Algorithm 1.

Defining $\mathcal{M}_a$ as the model that undergoes adversarial training during the training process. Adversarial training can be defined as a min-max optimization problem, where the model aims to minimize the expected loss on the entire dataset, even in case of encountering adversarial samples with maximum disturbance [24, 41]. It can be formulated as follows:

$$\min_{\theta} \mathbb{E}_{(X,y)\sim\mathcal{D}}[max\mathcal{L}(\mathcal{M}_\theta(\mathcal{A}(X)), y)] \quad (1)$$

where X and y are the input data and its corresponding label sampled from dataset $\mathcal{D}$, $\mathcal{L}$ is the loss function, $\mathcal{M}_\theta$ is the target model with parameter θ and $\mathcal{A}$ is the way to add disturbance to the input X. In MARVEL, $\mathcal{A}$ is defined as adding noise on the continuous

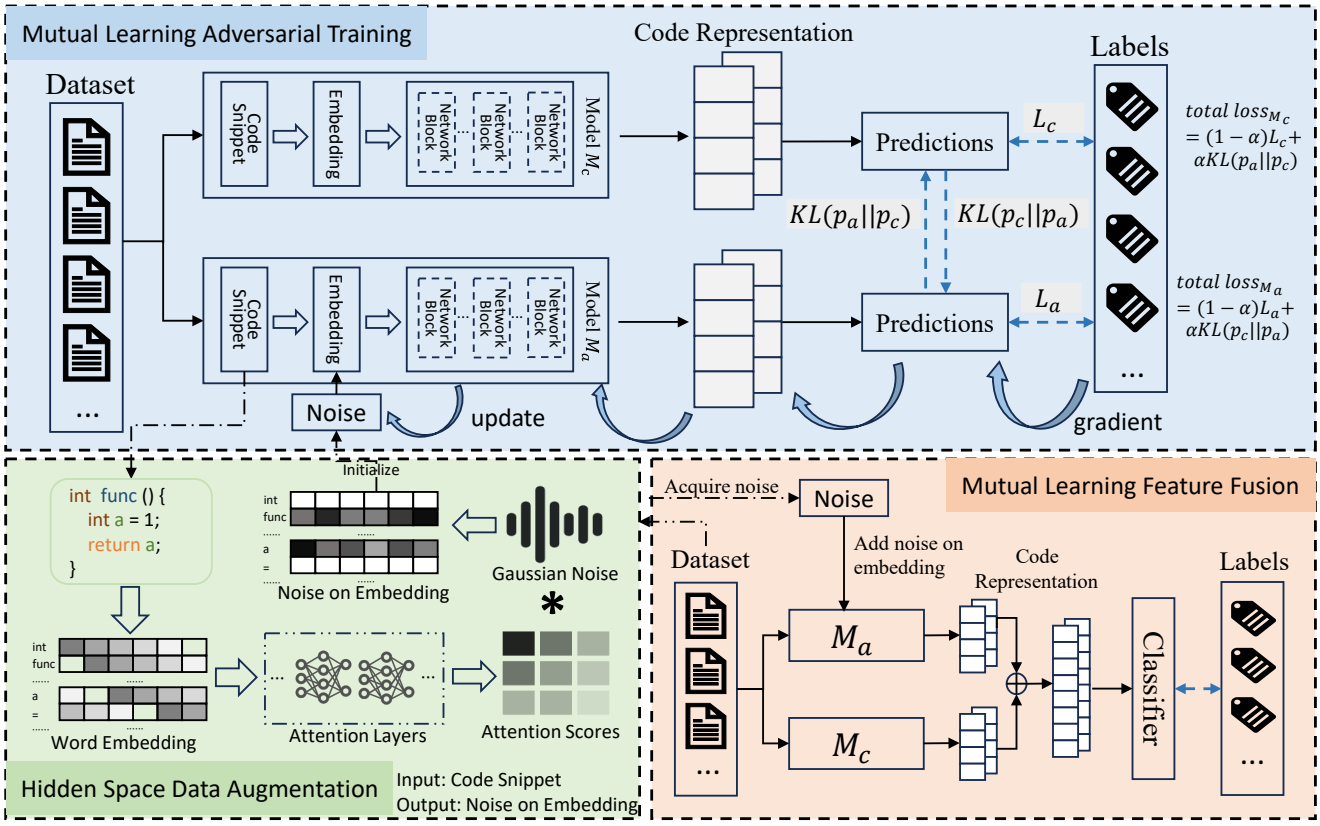

Figure 2: The overall workflow of MARVEL.

embedding space, we first generate initial noise which has been provided in section 3.2, and then update the noise during the adversarial training. So the adversarial training target can be defined as follows:

$$\min_{\theta} \mathbb{E}_{(X,y)\sim\mathcal{D}}\Big[\max_{||\delta||_F\leq\epsilon} \mathcal{L}(\mathcal{M}_\theta(\mathcal{T}(X)+\delta), y)\Big] \tag{2}$$

where $\mathcal{T}$ the encoder that transforms the input X into an embedding vector. For every single input, we are supposed to find the optimal $\delta \in \mathcal{R}^{1\times d}$ to maximize the inner part of Eq. 2, which is a non-concave optimization problem. Inspired by the Project Gradient Descent (PGD)[41] which has been proven effective in addressing this issue, MARVEL uses a gradient-driven approach to adaptively update the initially generated noise δ based on the model's internal parameter information. This process can be represented as:

$$\begin{aligned} g(\delta_t) &= \nabla_{\delta_t}\mathcal{L}(\mathcal{M}_a(\mathcal{T}(X+\delta_t)), y) \\ \delta_{t+1} &= \Pi_{||\delta||_F\leq\epsilon}\big(\delta_t + \mu\frac{g(\delta_t)}{||g(\delta_t)||_F}\big) \end{aligned} \tag{3}$$

where Π is a projecting function that constrain δ within the ϵ-ball, ∇ is a function that compute the gradient based on the loss and t represents the number of iterative computation steps, i.e. during the adversarial training process, the gradient will be repeatedly calculated and the noise δ will be updated. This process will be repeated K times, where K is a hyper-parameter representing the number of steps for gradient accumulation.

The purpose of adversarial training is to enable the model to learn the capability of being resistant to disturbances. However, solely through this process, the model can easily suffer from reduced accuracy. Therefore, we introduce DML, setting up another model $\mathcal{M}_c$ that only receives original clean data, let $\mathcal{M}_c$ and $\mathcal{M}_a$ learn from each other. Our basic idea is that $\mathcal{M}_a$ can learn correct classification capability from $\mathcal{M}_c$, while $\mathcal{M}_c$ can learn the robustness against disturbances from $\mathcal{M}_a$. We explain our approach using a simple classification problem, in the training process of a single model, the objective function is defined as the cross entropy error between the true label and the predicted probability distribution:

$$L_{\mathcal{M}} = -\sum_{i=1}^{N}\sum_{m=1}^{M} I(y_i, m) log(p_{\mathcal{M}}^m(x_i)) \tag{4}$$

where $p_{\mathcal{M}}^m(x_i)$ represents the probability of the i-th data on the m-th classification by the model $\mathcal{M}$ and I is an indicator function defined as follows:

$$I(y_i, m) = \begin{cases} 1 & y_i = m \\ 0 & y_i \neq m \end{cases} \tag{5}$$

Both $\mathcal{M}_a$ and $\mathcal{M}_c$ contain this objective function in their training process. In order to enable $\mathcal{M}_c$ to learn the robustness capability from $\mathcal{M}_a$, DML allows $\mathcal{M}_a$ to provide the probability distribution of $\mathcal{M}_a$ on the input data as training experience for $\mathcal{M}_c$. Following [66], we use the KL Divergence to measure the difference between

Algorithm 1 Mutual Learning Adversarial Training

Inputs:
$\mathcal{D} = \{(x, y)\}$: training sets;
η: learning rate;
K: ascent steps;
μ: adversarial learning rate;
μ: training epoch;
Outputs:
Θ_a: model $\mathcal{M}_a$'s parameters;
Θ_c: model $\mathcal{M}_c$'s parameters

for $epoch = 1, 2, ..., N$ **do**
  for mini-batch $\mathcal{B} \subset \mathcal{D}$ **do**
    for $i = 1, 2, ..., m$ **do**
      Initial noise δ_i
    end for
    for $t = 1$ to K **do**
      $p_a \leftarrow \mathcal{M}_a(x + \delta)$, $p_c \leftarrow \mathcal{M}_a(x)$
      $\Theta_a \leftarrow \Theta_a + \eta \frac{\partial \mathcal{L}_{\Theta_a}}{\partial \Theta_a}$
      for $i = 1, 2, ..., m$ **do**
        $g_i \leftarrow \nabla \mathcal{L}(\mathcal{M}_a(x_i + \delta), y_i)$
        $\delta_i \leftarrow \delta_i + \mu \frac{g_i}{||g_i||_F}$
      end for
    end for
    $\Theta_a \leftarrow \Theta_a + \eta \frac{\partial \mathcal{L}_{\Theta_a}}{\partial \Theta_a}$
    $p_a \leftarrow \mathcal{M}_a(x + \delta)$, $p_c \leftarrow \mathcal{M}_a(x)$
    $\Theta_c \leftarrow \Theta_c + \eta \frac{\partial \mathcal{L}_{\Theta_c}}{\partial \Theta_c}$
  end for
end for

the probability distributions of the two models:

$$D_{KL}(p_{\mathcal{M}_a}||p_{\mathcal{M}_c}) = \sum_{i=1}^{N}\sum_{m=1}^{M} p^m_{\mathcal{M}_a}(x_i) log \frac{p^m_{\mathcal{M}_a}(x_i)}{p^m_{\mathcal{M}_c}(x_i)} \tag{6}$$

We also hope that $\mathcal{M}_a$ can learn the correct classification capability from $\mathcal{M}_c$, so $\mathcal{M}_c$ also needs to provide its probability distribution as learning experience for $\mathcal{M}_a$. Ultimately, the loss functions of $\mathcal{M}_a$ and $\mathcal{M}_c$ will be respectively defined as:

$$\begin{aligned} \mathcal{L}_{\mathcal{M}_a} &= (1-\alpha)L_{\mathcal{M}_a} + \alpha D_{KL}(p_{\mathcal{M}_c}||p_{\mathcal{M}_a}) \\ \mathcal{L}_{\mathcal{M}_c} &= (1-\alpha)L_{\mathcal{M}_c} + \alpha D_{KL}(p_{\mathcal{M}_a}||p_{\mathcal{M}_c}) \end{aligned} \tag{7}$$

Where α is a hyper-parameter used to control the strength of mutual learning. As α increases, the weight of the mutual learning component in the loss function becomes higher, leading to stronger mutual learning.

For each training batch, $\mathcal{M}_a$ will first receive the data with the initialized noise added, and update the noise based on the gradient. Afterwards, both $\mathcal{M}_a$ and $\mathcal{M}_c$ will update their output probability distributions, and update their model parameters according to the loss functions defined above.

Since $\mathcal{M}_a$ receives data with noise add to the identifiers, this effectively weakens the influence of the original identifier names on the results, making the model no longer classify the input based on the identifiers names. Therefore, to maintain this effect and data consistency, unlike general adversarial training, we add noise to input data of $\mathcal{M}_a$ in both the training and inference stages, but only use gradient to update the noise during the training stage. In contrast, $\mathcal{M}_c$ only receives clean data.

3.4 Mutual Learning Feature Fusion

After completing MutAT and obtaining the two robust models $\mathcal{M}_c$ and $\mathcal{M}_a$, both of these models have learned robust code representations through the MutAT process. In order to better integrate the performance of the two models and apply it to specific downstream tasks, we design MutFF to fuse the code representations obtained from the two models.

Our intuition is to simultaneously consider the code features under the two different input modes and have them jointly participate in the model's output decision. Due to the fact that model $\mathcal{M}_a$ received data with added noise during the MutAT process, its learned code representation has better robustness. On the other hand, model $\mathcal{M}_c$ received clean data, so its code representation has better accuracy. In the code tasks presented in this paper, all code representations need to be fitted to a classifier with the true labels. Therefore, a method that can leverage both types of code representations is to concatenate the two code representations and then feed them into the classifier, in order to enable the model to capture information from both dimensions when facing the same input.

Specifically, MutFF takes the last layer hidden state distributions $S_{\mathcal{M}_a}$ and $S_{\mathcal{M}_c}$ from $\mathcal{M}_a$ and $\mathcal{M}_c$, i.e., code representation in Figure 2, directly concatenates them to form S', this process can be defined as:

$$S' = [S_{\mathcal{M}_c}, S_{\mathcal{M}_a}] \in \mathcal{R}^{2\times d} \tag{8}$$

where $[,]$ represents the vector concatenation operation and d represents the dimension of code representation. And MutFF sets up a classifier C' with the same structure as the model $\mathcal{M}_a$ and $\mathcal{M}_c$'s classifier but with a different input dimension, in order to recalculate the probability distribution as $p' = C'(S')$. In this process, $\mathcal{M}_a$ also needs to receive data with added noise, which is calculated from 3.2, and no longer update based on the gradient. Additionally, the model parameters of $\mathcal{M}_a$ and $\mathcal{M}_c$ need to be frozen as they have already been trained in section 3.3, and only the final C' is trained separately on the training set. The final C', $\mathcal{M}_a$ and $\mathcal{M}_c$ are then used as the ultimate model obtained by MARVEL.

4 EXPERIMENTAL SETUP

In this paper, we evaluate MARVEL by answering the following research questions (RQs):

RQ1: Does MARVEL improve the robustness of existing DCMs and how does MARVEL affect their accuracy?
RQ2: How do the different modules in MARVEL impact its overall performance?
RQ3: How do different parameter settings affect the performance of MARVEL?

4.1 Subjects

To provide a sufficient evaluation of our method, we employed five widely used datasets that encompass three popular programming languages (C, Java, and Python), covering four different code classification tasks (Authorship Attribution, Vulnerability Detection,

Table 1: Statistics of ours used datasets and models

Dataset	Train/Val/Test	Class	Language	Model	Acc.
Authorship Attribution	528/-/132	66	Python	CodeBERT	81.81%
				GCBERT*	77.27%
				UniXCoder	86.36%
Vulnerability Detection	21,854/2,732/2,732	2	C	CodeBERT	64.39%
				GCBERT	62.15%
				UniXCoder	65.74%
Defect Prediction	27,058/-/6,764	4	C/C++	CodeBERT	83.06%
				GCBERT	81.34%
				UniXCoder	85.85%
FC-Java250**	48,000/11,909/15,000	250	Java	CodeBERT	97.06%
				GCBERT	97.95%
				UniXCoder	98.13%
FC-Python800	153,600/38,400/48,000	800	Python	CodeBERT	97.93%
				GCBERT	98.52%
				UniXCoder	98.54%

* GCBERT is short for GraphCodeBERT.
** FC is short for Functionality Classification.

Defect Prediction and Functionality Classification) in our evaluation.

Authorship attribution aims to correctly identify the author of a given code snippet. We use the Google Code Jam (GCJ) dataset [6], which collects the code submitted by different participants in the *Google Code Jam* challenge, the dataset contains code snippets and their corresponding author IDs, with a total of 660 Python files submitted by 66 authors. We divided the dataset into an 80% training set and a 20% testing set following prior work [49, 56]. **Vulnerability detection** is a binary classification task to identify whether a code snippet contains vulnerabilities or not. We use the dataset that was proposed by Zhou et al. which contains two C projects [67]. This dataset is also part of the CodeXGLUE [40] benchmark, we follow the CodeXGLUE division of the training, validation, and test sets for this dataset. **Defect prediction** aims to predict whether a code snippet has defects and identify the specific type of defect it exhibits. We use the CodeChef dataset [1] following existing works [49, 62], which contains 33,822 C/C++ codes, and categorized them into four classes: OK, Wrong Answer, Time Limit Exceeded, and Running Error. **Functionality classification** aims to classify given code snippets based on the problems that they can solve. We use the CodeNet dataset [46] proposed by IBM for this task, which is a large-scale code dataset containing 14 million programming projects across 55 programming languages. Specifically, we selected the Java250 and Python800 subsets from CodeNet for the experiments, which consist 250 classes of Java code and 800 classes of Python code, respectively.

As for the code models, we employed three state-of-the-art pre-trained DCMs, i.e., CodeBERT [20], GraphCodeBERT [27], and UniXCoder [26], which are mentioned in Section 2.1. Table 1 shows the statistics of our used datasets and models, where Acc. represents the accuracy of the models after fine-tuning on the corresponding datasets, i.e., without applying any robustness improvement methods.

4.2 Measurements and Baselines

Our work primarily focuses on enhancing the robustness of DCMs, specifically their ability to withstand adversarial attacks. Therefore, we use state-of-the-art attack techniques to attack DCMs enhanced by MARVEL instead of creating a transformed test set for each dataset like the previous works [16, 21, 61]. So we chose the widely-used ASR [49, 56] as our evaluation metric. Given a victim code model $\mathcal{M}$ and a dataset $\mathcal{X}$, where each element $x \in \mathcal{X}$ is a code snippet that could be correctly classified by $\mathcal{M}$ and has at least one local variable, which means attackers can leverage attack methods that involve substituting variable names, then the ASR of an attack technique can be defined as $\frac{\{|x|x\in\mathcal{X}\wedge\mathcal{M}(x')\neq\mathcal{M}(x)\}|}{|\mathcal{X}|}$, where x' is the adversarial example generated by attackers. A lower ASR indicates that a model has better robustness.

In our study, we use three state-of-the-art attack techniques ALERT [56], MHM [63] and CODA [49] as they have demonstrated high ASR in attacking the current pre-trained DCMs. By evaluating the reduction in ASR, we can determine the effectiveness of our method in improving DCM's robustness.

We use CREAM [21] and SPACE [36] as our baselines. CREAM uses counterfactual reasoning framework to eliminate misleading information of identifiers to improve model robustness and have demonstrated excellent efficacy, while SPACE is able to boost the robustness of DCMs via a semantic-preserving adversarial training. Both CREAM and SPACE have been proven to be effective in improving the robustness of DCMs. However, GraphCodeBERT incorporates DFG as part of its input, while CREAM splits the input code into sequences containing only identifiers and code sequences without identifiers, resulting in the inability to capture the DFG. Hence, we do not conduct experiments using CREAM on GraphCodeBERT.

4.3 Implementations

We implemented MARVEL in Python and utilized tree-sitter [2] to extract identifiers from code snippets. We set the parameters $\alpha = 0.3$ and $K = 3$ in MARVEL by conducting a preliminary experiment, and we discuss the influence of these parameters in Section 5.3. The experiments related to CodeBERT and GraphCodeBERT were conducted on an Ubuntu 16.04 server with Intel(R) Xeon(R) E5-2683 v4 @2.10GHz CPU, and NVIDIA TITAN RTX GPU, and the experiments related to UniXCoder were conducted on an Ubuntu 20.04 server with Intel(R) Xeon(R) Platinum 8352V @2.10GHz CPU, and NVIDIA RTX 4090 GPU.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Evaluation on the Robustness of MARVEL

We applied three state-of-the-art adversarial attack algorithms (i.e., ALERT, MHM, and CODA) on the test sets of five datasets for three DCMs (i.e., CodeBERT, GraphCodeBERT and UniXCoder) that incorporated the MARVEL. We also compared the MARVEL with two baselines, CREAM and SPACE. The reduction in the ASR of these attack methods serves as a quantitative indicator to measure their robustness, a lower ASR indicates better robustness. The full results illustrated in Table 2. Based on these results, we have the following observations: **Firstly, MARVEL can significantly improve the robustness of DCMs across all 45 tasks.** Specifically, the original CodeBERT, GraphCodeBERT, and UniXCoder achieved an average

Table 2: Comparison results of ASR results of state-of-the-art adversarial attacks on attacking three models after applying MARVEL, CREAM, and SPACE. (AA is short for Authorship Attribution, VD is short for Vulnerability Detection, and DP is short for Defect Prediction)

Approach	AA			VD			DP			FC-Java250			FC-Python800		
	ALERT	MHM	CODA	ALERT	MHM	CODA	ALERT	MHM	CODA	ALERT	MHM	CODA	ALERT	MHM	CODA
CodeBERT	37.96%	68.52%	84.91%	58.70%	71.11%	97.78%	73.34%	47.99%	68.24%	33.96%	39.41%	59.45%	54.28%	50.05%	37.76%
+CREAM	26.89%	55.46%	41.88%	56.76%	73.18%	95.49%	62.51%	45.39%	54.59%	16.03%	29.60%	38.98%	26.02%	**31.33%**	19.83%
+SPACE	37.84%	70.27%	84.40%	61.58%	84.86%	96.99%	75.50%	68.97%	61.94%	30.83%	33.10%	55.21%	53.22%	48.33%	40.48%
+MARVEL	**9.17%**	**12.04%**	**13.08%**	**30.01%**	**69.88%**	**80.17%**	**38.13%**	**45.38%**	**54.35%**	**11.28%**	**17.68%**	**10.99%**	**13.95%**	35.08%	**11.77%**
GCBERT	61.76%	75.49%	92.86%	80.58%	87.40%	98.52%	83.12%	64.81%	58.59%	35.82%	26.76%	53.82%	54.78%	46.25%	34.32%
+SPACE	76.47%	83.33%	90.00%	75.96%	84.34%	96.54%	83.18%	70.67%	60.53%	31.83%	25.59%	52.35%	56.91%	55.02%	41.11%
+MARVEL	**28.57%**	**34.29%**	**41.75%**	**49.28%**	**68.32%**	**87.56%**	**51.80%**	**45.83%**	**35.45%**	**13.42%**	**3.67%**	**5.71%**	**22.79%**	**17.49%**	**7.73%**
UniXCoder	45.61%	55.31%	69.64%	63.84%	80.23%	91.93%	74.92%	54.71%	65.19%	28.60%	38.79%	54.89%	45.51%	39.12%	28.62%
+CREAM	30.51%	51.69%	51.72%	44.58%	75.86%	95.34%	50.44%	53.35%	**49.67%**	12.29%	18.28%	17.93%	31.56%	30.98%	17.56%
+SPACE	63.63%	77.27%	92.59%	63.14%	86.55%	97.03%	84.89%	76.98%	66.65%	27.45%	33.65%	52.89%	47.25%	41.18%	31.13%
+MARVEL	**11.50%**	**15.18%**	**14.16%**	**44.24%**	**66.53%**	**87.57%**	**34.88%**	**46.17%**	50.67%	**9.94%**	**5.58%**	**4.46%**	**11.67%**	**20.27%**	**5.40%**

Table 3: Comparison results of the test set accuracy for the three models after applying MARVEL, CREAM, and SPACE.

Approach	AA	VD	DP	Java250	Python800
CodeBERT	81.82%	64.39%	83.06%	97.06%	97.93%
+CREAM	**90.15%**	**64.60%**	84.06%	**99.14%**	**98.95%**
+SPACE	84.85%	64.57%	81.99%	98.51%	98.60%
+MARVEL	82.58%	63.18%	**84.27%**	95.59%	96.79%
GCBERT	77.27%	62.15%	81.34%	97.95%	98.52%
+SPACE	78.79%	**64.24%**	82.69%	**97.95%**	**98.72%**
+MARVEL	**80.30%**	62.66%	**83.59%**	97.69%	98.25%
UniXCoder	86.36%	**65.74%**	85.85%	98.13%	98.54%
+CREAM	**89.39%**	64.82%	84.57%	**98.83%**	98.91%
+SPACE	83.33%	65.30%	82.87%	98.61%	**99.02%**
+MARVEL	86.36%	65.30%	**86.62%**	97.23%	97.92%

ASR of 58.90%, 63.65%, and 55.79% across three datasets and three attack algorithms. After being strengthened by MARVEL, their average ASR was reduced to 30.20%, 34.24%, and 28.55%, respectively, with a decrease of 28.70%, 29.41%, and 27.24%.

Secondly, the robustness improvement of MARVEL surpasses that of the two baselines. Specifically, the average ASR for CREAM-enhanced CodeBERT and UniXCoder are 44.93% and 42.12% respectively in the 30 related tasks of CREAM. Our MARVEL outperforms CREAM in the 28 tasks, in which reducing average ASR by 16.05%, and 14.61%. Additionally, the average ASR for SPACE-enhanced CodeBERT, GraphCodeBERT and UniXCoder are 60.23% , 65.59%, and 62.82% respectively in the all 45 tasks. Our MARVEL also surpasses SPACE in all 45 tasks, with ASR reduced by 30.04%, 31.34%, and 34.27%.

In addition, to examine the impact of MARVEL on the intrinsic performance of the models, we also evaluated the accuracy of the MARVEL-based models on the original test sets, with the results illustrated in Table 3. Based on these results, we can conclude that **MARVEL can effectively maintain the intrinsic performance of DCMs, and even improve the model performance on certain tasks.** For example, on the DP task, MARVEL boosted the performance of CodeBERT, GraphCodeBERT, and UniXCoder by 1.19%, 2.25%, and 0.77% respectively, even outperforming the two baselines CREAM and SPACE. For the AA task, although the performance improvement capability of MARVEL is weaker compared to CREAM and SPACE, it still achieved performance gains relative to the original CodeBERT and GraphCodeBERT models. However, for the Java250 and Python800 datasets, we unfortunately found that MARVEL caused a certain degree of performance loss. For example, on the Java250 dataset, MARVEL led to 1.47%, 0.26%, and 0.9% losses for CodeBERT, GraphCodeBERT, and UniXCoder respectively. This may be because the Java250 and Python800 datasets are larger, and the limited number of iteration rounds is not enough to support MARVEL achieving better results. Nevertheless, MARVEL can achieve substantial robustness improvements at the cost of only minor performance loss, which we believe is a more critical aspect in the practical application of DCMs.

Answer to RQ1: MARVEL can significantly improve the robustness of DCMs across all 45 tasks while maintaing the intrinsic performance of the models. Compared to the baselines, MARVEL achieved the best performance on 43 out of the 45 tasks, further reducing ASR by an additional 15.33% and 31.88%.

5.2 RQ2: Ablation Study

We further conducted ablation studies to validate the effectiveness of the three key modules in MARVEL that are fundamentally different from existing work, i.e., HidDA, MutAT and MutFF. We selected the AA dataset to experiment on CodeBERT, GraphCodeBERT, and UniXCoder, in order to examine the impact of different variants

Table 4: Ablation Study Results on AA Dataset.

Variants	CodeBERT				GraphCodeBERT				UniXCoder			
	Accuracy	ALERT	MHM	CODA	Accuracy	ALERT	MHM	CODA	Accuracy	ALERT	MHM	CODA
Original	81.82%	37.96%	68.52%	84.91%	77.27%	61.76%	75.49%	92.86%	86.36%	45.61%	53.51%	69.64%
+MARVEL	**82.58%**	9.17%	**12.04%**	**13.08%**	80.30%	**28.57%**	**34.29%**	41.75%	86.36%	**11.50%**	15.18%	**14.16%**
-w/o mutual	78.79%	20.00%	16.19%	13.86%	68.94%	40.45%	42.05%	43.33%	79.55%	64.86%	51.35%	44.24%
w/ both noise	77.27%	**7.55%**	18.87%	13.46%	71.97%	38.95%	51.58%	44.08%	80.30%	12.15%	**14.95%**	14.29%
w/ both clean	80.30%	41.18%	67.65%	73.00%	**82.58%**	69.72%	82.57%	88.79%	**87.12%**	85.71%	93.88%	88.42%
-w/o contact-$\mathcal{M}_a$	81.06%	8.41%	15.89%	14.29%	74.24%	33.33%	41.84%	**32.98%**	81.06%	18.87%	21.15%	16.35%
-w/o contact-$\mathcal{M}_c$	82.58%	15.60%	28.44%	39.25%	82.58%	50.46%	68.81%	80.37%	87.12%	24.35%	33.04%	38.94%

on the model performance and their robustness against the three attack methods. The experimental results are presented in Table 4.

1) HidDA and MutAT: MARVEL includes a mutual learning-based adversarial training process, where one model receives clean data while the other receives noisy data for hidden space data augmentation and performs adversarial training. Simultaneously, the two models learn from each other. To investigate the effectiveness of the two modules, we designed three variants:

- **w/o mutual:** we kept only one model during the training process and removed the mutual learning module, essentially training just a single model with adversarial training on the embedding.
- **w/ both noise:** both models in the mutual learning process received noisy data and underwent adversarial training.
- **w/ both clean:** both models in the mutual learning process received only clean data, without HidDA and any adversarial training, essentially a pure mutual learning setup.

From Table 4, we can observe that these three variants all suffered from different degrees of performance degradation. Specifically, compared to MARVEL, the w/o mutual variant incurred a loss in the model's original accuracy, and its improvement in the model's robustness was also weaker. For example, for the CodeBERT model, the w/o mutual variant saw a 0.76% decrease in accuracy, and the ASR against ALERT, MHM, and CODA increased by 10.83%, 4.15%, and 0.78% respectively, compared to MARVEL. As for the w/ both noise variant, it can achieve improvements in the robustness compared to the original model, and in some cases even perform better than MARVEL (e.g., compared to MARVEL, the ASR for the MHM algorithm on UniXCoder was further reduced by 0.23%). This is because both models receive noisy data, allowing them to simultaneously obtain good robustness and more effectively fit the noisy data. However, due to the lack of learning from clean data, it also suffers from severe accuracy degradation in almost all cases. The w/ both clean variant, on the other hand, can maintain the model's accuracy and even improve it on some models (e.g., 5.31% and 0.76% increases for GraphCodeBERT and UniXCoder respectively compared to the original). However, its improvement in the robustness is very marginal. These results demonstrate that compared to adversarial training alone, the approach of combining mutual learning can effectively improve the model's robustness. By setting different inputs for the two models, where one receives clean data and the other receives noisy data, the two model can learn to fit clean data and defend against noise, respectively. This allows the model to withstand adversarial attacks while maintaining its own accuracy.

2) MutFF: In Section 3.3, we had obtained two models $\mathcal{M}_a$ and $\mathcal{M}_c$, and we extracted the code representations from these two models and concatenated them, feeding the concatenated representation into a classification model in Section 3.4. To validate the effectiveness of this module, we removed MutFF and designed two variants: **w/o contact-$\mathcal{M}_a$** and **w/o contact-$\mathcal{M}_c$**. Essentially, these are the $\mathcal{M}_a$ and $\mathcal{M}_c$ obtained in Section 3.3, but we fed their respective code representations directly into the classifier to obtain the classification results, without concatenating them.

From the results in Table 4, we can observe that the w/o contact-$\mathcal{M}_a$ variant has relatively better robustness, but with some accuracy loss. On the other hand, the w/o contact-$\mathcal{M}_c$ variant can maintain the model's original accuracy, but its robustness is slightly weaker than w/o contact-$\mathcal{M}_a$. The MARVEL, by concatenating the representations from these two models, solves this trade-off. MARVEL harnesses the advantages of both models, it can synergistically leverage $\mathcal{M}_c$'s capability to fit clean data and $\mathcal{M}_a$'s robustness to input perturbations.

Answer to RQ2: The different modules in MARVEL, i.e., HidDA, MutAT, and MutFF, all have a positive impact on MARVEL's overall performance, and the rationality of their design has been demonstrated.

5.3 RQ3: Evaluation on the Parameter Settings

We analyzed the impact of two key hyper-parameters, K and α introduced in Section 3.3, of MARVEL on the model's accuracy and robustness, as these two parameters have the most significant influence on the results. We conducted experiments using the AA dataset as an example. The results are shown in Figure 3.

The parameter K. The parameter K determines the number of gradient update steps for the noise, where K=1 means that the noise is not updated according to the gradient. A larger K means a higher intensity of added noise and more adversarial training. We set $K = \{1, 2, 3, 4, 5\}$ to investigate the impact of different values of K on the results. As shown in Figure 3(g), 3(b) and 3(c), we can observe that for all tasks and models, the capability to withstand adversarial attacks when $K > 1$ is stronger than when $K = 1$. That

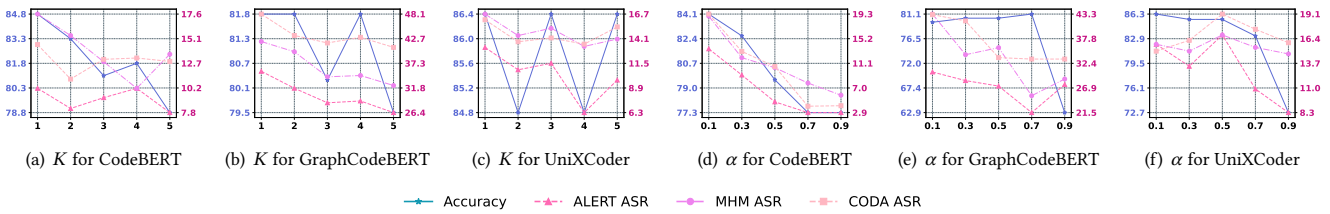

Figure 3: Hyper-Parameter Analysis. The left and right vertical axes represent the model's accuracy on the test set and the ASR of the three attack algorithms, respectively.

is, the gradient-driven noise update strategy can effectively improve the model's robustness. We also notice that as the K value increases, it may lead to a loss in model accuracy. This is likely because as the number of noise updates increases, the model's decision boundary expands, making the model more prone to overfitting to the noise, which in turn reduces the model's generalization capability.

The result shown in Figure 3 demonstrates our experimental results on the AA dataset. Based on these results, we can observe that setting $K = 3$ or $K = 4$ can exhibit better performance. However, MARVEL is a general framework that can be applied to different code models and datasets. After evaluating all five datasets and three models, we determine that generally $K = 3$ is the optimal choice. Therefore, we set $K = 3$ as the default setting for our MARVEL framework.

The hyper-parameter α. The hyper-parameter α represents the strength of the mutual learning process, which determines how much information the model obtains from the mutual learning process. We experimented with $\alpha = \{0.1, 0.3, 0.5, 0.7, 0.9\}$, as shown in Figure 3(d), 3(e) and 3(f), as α increases, the model's robustness gradually improves, but its inherent accuracy also decreases. A larger α means a stronger mutual learning process, with a smaller proportion of fitting the original labels. Through the mutual learning process, the model learns more noise-resistant capabilities, but this also weakens its performance on fitting the original data. In this work, we set $\alpha = 0.3$, as it can improve the model's robustness while reasonably preserving its accuracy.

Answer to RQ3: As K and α increase, MARVEL's robustness against adversarial attacks is enhanced, but it results in a slight loss of accuracy. We balanced the model's robustness and accuracy to determine the default settings for these hyper-parameters.

6 DISCUSSION

6.1 The Time and Memory Cost

Due to the fact that the training stage of MARVEL requires simultaneous updates of the parameters for the two models, as well as gradient-based iterative updates of the noise, and the inference stage of MARVEL also necessitates obtaining the code representations from the two models and concatenating them, MARVEL incurs certain time and memory cost compared to not applying MARVEL (i.e., only fine-tuning on the dataset).

In our experiments, the training and inference time of MARVEL were about 4 times longer compared to only fine-tuning, while the memory overhead was around twice that of fine-tuning alone. As the value of K increases, the number of iterations for updating the noise also increases, leading to longer training time. Taking the results of CodeBERT on the AA dataset as an example, the training time when K was set to 1, 3, 5 was 3.47, 5.7, and 8 times longer than the time required for only fine-tuning, respectively. We traded off some time and memory overhead to achieve better model robustness. In the future, we will explore more efficient ways to improve model robustness.

6.2 Threats to Validity

The main threat to the *internal* validity lies in the parameter settings of MARVEL. In this work, we studied the two most critical parameters (i.e., K and α as described in Section 5.3). Figure 3 show the impact of K and α on the performance. To balance the model's robustness and accuracy, we set K to 3 and α to 0.3 as the default configuration for MARVEL. Another internal threat is the implementation of MARVEL. To mitigate this threat, we carefully conducted code reviews and released artifacts that can be replicated and used in practice.

The main threat to the *construction* validity lies in the measurements used in the experiments. To mitigate this threat, we employ evaluation metrics that have been widely used in prior works, i.e., accuracy and attack success rate.

The main threat to the *external* validity lies in the subjects that we selected for the experiments. To mitigate this threat, we used three different DCMs, evaluated them on five widely-used datasets covering three popular programming languages, and employed three attack algorithms to assess the robustness. Besides, we have extended the two baselines to the models and datasets used in this work, and the correctness of this extension may also pose a potential threat to the validity.

7 RELATED WORK

To evaluate and improve the performance and robustness of DCMs, researchers have made many efforts, which can be categorized into two types: generating adversarial examples to reveal the defects of DCMs, and proposing advanced frameworks to improve the performance and robustness of the models.

7.1 Adversarial Attack on Code

Deep learning systems have been proven to be vulnerable to adversarial attacks[12, 15, 18, 22, 23]. Adversarial attacks aim to generate adversarial examples by adding slight perturbations to the original inputs of DNNs to mislead them into producing incorrect outputs.

Recently researchers proposed many techniques to generate adversarial examples specifically targeting DCMs [49, 56, 62–64]. One purpose of conducting adversarial attacks is that the generated adversarial samples can be added to the training set to retrain the model, in order to improve the robustness of the model (at the data-level).

There are many adversarial example generation techniques for DCMs. Yefet et al. proposed DAMP [57] that changes variable names using gradient information. Zhang et al. proposed MHM [63], which is a Metropolis-Hastings sampling-based identifier renaming technique. Yang et al. proposed ALERT [56], which generates substitutes that are aware of natural semantics and use genetic algorithms to perform variable replacement. Tian et al. proposed CODA [49] that can transform code structure and rename variables considering code structure differences and identifier differences. Zhang et al. proposed RNNS [64], which use representation nearest neighbor search to find potential adversarial substitutes. All these techniques have demonstrated strong attack capabilities against DCMs. In this paper, our purpose is to improve the model's ability to resist these adversarial attacks, and we employ MHM, ALERT and CODA as attack techniques to evaluate our approach.

7.2 Performance & Robustness on Code Models

To improve the performance or robustness of DCMs, researchers have also made great efforts. Besides the two baselines CREAM [21] and SPACE [36] that have already been mentioned in this paper, Dong et al. proposed MixCode [16], which could enhance code classification by mixup-based data augmentation. Tian et al. proposed CodeDenoise [50] that could on-the-fly improve code model's performance via input denoising. Li et al. proposed RoPGen [37] that could robust code authorship attribution via automatic coding style transformation. In addition, some works have used contrastive learning techniques to design new network architectures and loss functions to improve the performance of code modesl [31, 38, 51, 52].

However, there is still a lack of research that truly focuses on the robustness of DCMs against the perturbation of adversarial examples, i.e., DCM could maintain correct outputs even when faced with adversarial examples. The ability of the model to defend adversarial attacks means that the model has high robustness, which is of great significance in the application of real scenarios. Although researchers have proposed many techniques to counter adversarial attacks in other domains of deep learning, based on existing work [60], these techniques can be categorized into adversarial detecting [39, 43], input reconstruction [42, 48], network verification [25, 32], network distillation [45], adversarial training [24, 33] and classifier robustifying [8]. When it comes to DCMs, a common practice is to augment the training set by incorporating the generated adversarial examples and then retraining the model [49, 56, 62], which does not directly enhance the model's robustness at the model-level, and this approach makes it difficult to resist multiple types of adversarial attacks simultaneously.

Besides, existing works [16, 21, 31, 36–38, 50] primarily focus on improving the accuracy of DCMs, when it comes to measuring model robustness, they construct a transformed test set to evaluate the model's accuracy instead of utilizing state-of-the-art attack algorithms. While some studies [21, 36] has employed state-of-the-art attack algorithms such as ALERT to evaluate the robustness of DCMs by attacking the DCMs, there is still a lack of methods that can enhance the robustness to input perturbations across multiple downstream tasks for multiple DCMs. Additionally, there is a shortage of large-scale experimental validations in this area, which motivated us to proposed a more powerful robustness enhanced technique for DCMs, i.e., MARVEL.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose MARVEL, a mutual learning-based framework for enhancing robustness of code models via adversarial training that can be applied to any DCM that needs to be fine-tuned on a code dataset. MARVEL improves model robustness while maintaining accuracy through a deep mutual learning approach between two models. One model is trained on carefully designed noisy inputs, while the other model is trained on the original clean data. We conducted extensive experiments across 5 datasets and 3 models. The results demonstrate that, compared to the baselines, MARVEL can significantly reduce the ASR of state-of-the-art attack algorithms, while maintaining the inherent accuracy of the models. This validates MARVEL's ability to enhance the robustness of code models.

In the future, we will explore more efficient methods to improve the robustness of code models, reducing the time and memory costs. We will also conduct evaluations on a wider range of downstream tasks and datasets.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62372005 and 62322208.

REFERENCES

[1] 2023. "CodeChef". https://codechef.com/.
[2] 2023. "tree-sitter". https://tree-sitter.github.io/tree-sitter/.
[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.
[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. arXiv:1808.01400 [cs.LG]
[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
[6] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source code authorship attribution using long short-term memory based networks. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I 22*. Springer, 65–82.
[7] Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *International Conference on Machine Learning*. PMLR, 896–907.
[8] John Bradshaw, Alexander G de G Matthews, and Zoubin Ghahramani. 2017. Adversarial examples, uncertainty, and transfer testing robustness in Gaussian process hybrid deep networks. *arXiv preprint arXiv:1707.02476* (2017).
[9] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).
[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[11] Yizhou Chen, Zeyu Sun, Zhihao Gong, and Dan Hao. 2024. Improving Smart Contract Security with Contrastive Learning-based Vulnerability Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–11.

[12] Zhaoyu Chen, Bo Li, Shuang Wu, Kaixun Jiang, Shouhong Ding, and Wenqiang Zhang. 2024. Content-based unrestricted adversarial attack. *Advances in Neural Information Processing Systems* 36 (2024).

[13] Jinhao Dong, Yiling Lou, Dan Hao, and Lin Tan. 2023. Revisiting learning-based commit message generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 794–805.

[14] Jinhao Dong, Qihao Zhu, Zeyu Sun, Yiling Lou, and Dan Hao. 2023. Merge Conflict Resolution: Classification or Generation?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1652–1663.

[15] Yinpeng Dong, Zhijie Deng, Tianyu Pang, Jun Zhu, and Hang Su. 2020. Adversarial distributional training for robust deep learning. *Advances in Neural Information Processing Systems* 33 (2020), 8270–8283.

[16] Zeming Dong, Qiang Hu, Yuejun Guo, Maxime Cordy, Mike Papadakis, Zhenya Zhang, Yves Le Traon, and Jianjun Zhao. 2023. MixCode: Enhancing Code Classification by Mixup-Based Data Augmentation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 379–390.

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

[18] Ranjie Duan, Yuefeng Chen, Dantong Niu, Yun Yang, A Kai Qin, and Yuan He. 2021. Advdrop: Adversarial attack to dnns by dropping information. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 7506–7515.

[19] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 516–527.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[21] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. 2023. Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1933–1945.

[22] Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. 2020. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *Proceedings of the acm/ieee 42nd international conference on software engineering*. 1147–1158.

[23] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[24] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[25] Divya Gopinath, Guy Katz, Corina S Pasareanu, and Clark Barrett. 2017. Deepsafe: A data-driven approach for checking adversarial robustness in neural networks. *arXiv preprint arXiv:1710.00486* (2017).

[26] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225.

[27] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[28] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 526–537.

[29] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[30] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 293–304.

[31] Jinghan Jia, Shashank Srikant, Tamara Mitrovska, Chuang Gan, Shiyu Chang, Sijia Liu, and Una-May O'Reilly. 2023. CLAWSAT: Towards Both Robust and Accurate Code Models. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 212–223.

[32] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Towards proving the adversarial robustness of deep neural networks. *arXiv preprint arXiv:1709.02802* (2017).

[33] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236* (2016).

[34] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 249–260.

[35] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.

[36] Yiyang Li, Hongqiu Wu, and Hai Zhao. 2022. Semantic-Preserving Adversarial Code Comprehension. In *Proceedings of the 29th International Conference on Computational Linguistics*. 3017–3028.

[37] Zhen Li, Guenevere Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. Ropgen: Towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th International Conference on Software Engineering*. 1906–1918.

[38] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2476–2487.

[39] Jiajun Lu, Theerasit Issaranon, and David Forsyth. 2017. Safetynet: Detecting and rejecting adversarial examples robustly. In *Proceedings of the IEEE international conference on computer vision*. 446–454.

[40] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[41] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).

[42] Dongyu Meng and Hao Chen. 2017. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 135–147.

[43] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. 2017. On detecting adversarial perturbations. *arXiv preprint arXiv:1702.04267* (2017).

[44] CheolWon Na, YunSeok Choi, and Jee-Hyong Lee. 2023. 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)*. 7777–7791.

[45] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 582–597.

[46] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021).

[47] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[48] Yang Song, Taesup Kim, Sebastian Nowozin, Stefano Ermon, and Nate Kushman. 2018. PixelDefend: Leveraging Generative Models to Understand and Defend against Adversarial Examples. In *International Conference on Learning Representations*.

[49] Zhao Tian, Junjie Chen, and Zhi Jin. 2023. Code Difference Guided Adversarial Example Generation for Deep Code Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 850–862.

[50] Zhao Tian, Junjie Chen, and Xiangyu Zhang. 2023. On-the-fly Improving Performance of Deep Code Models via Input Denoising. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 560–572.

[51] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556* (2021).

[52] Xiao Wang, Qiong Wu, Hongyu Zhang, Chen Lyu, Xue Jiang, Zhuoran Zheng, Lei Lyu, and Songlin Hu. 2022. Heloc: Hierarchical contrastive learning of source code representation. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 354–365.

[53] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1069–1088.

[54] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[55] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 87–98.

[56] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*. 1482–1493.

[57] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[58] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 163–174.

[59] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data augmentation by program transformation. *Journal of Systems and Software* 190 (2022), 111304.

[60] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2019. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems* 30, 9 (2019), 2805–2824.

[61] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 39–51.

[62] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards robustness of deep program processing models—detection, estimation, and enhancement. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–40.

[63] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1169–1176.

[64] Jie Zhang, Wei Ma, Qiang Hu, Shangqing Liu, Xiaofei Xie, Yves Le Traon, and Yang Liu. 2023. A Black-Box Attack on Code Models via Representation Nearest Neighbor Search. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 9706–9716.

[65] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[66] Ying Zhang, Tao Xiang, Timothy M Hospedales, and Huchuan Lu. 2018. Deep mutual learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4320–4328.

[67] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

[68] Qihao Zhu, Qingyuan Liang, Zeyu Sun, Yingfei Xiong, Lu Zhang, and Shengyu Cheng. 2024. GrammarT5: Grammar-Integrated Pretrained Encoder-Decoder Neural Model for Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[69] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2021. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–31.