5-2023

ContraBERT: Enhancing code pre-trained models via contrastive learning

Shangqing LIU

Bozhi WU

Xiaofei XIE
Singapore Management University, xfxie@smu.edu.sg

Guozhu MENG

Yang. LIU

ContraBERT: Enhancing Code Pre-trained Models via Contrastive Learning

Shangqing Liu[1], Bozhi Wu[1], Xiaofei Xie[2†], Guozhu Meng [3], and Yang Liu[1]
[1]Nanyang Technological University, Singapore
[2]Singapore Management University, Singapore
[3]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
{shangqin001,bozhi001}@e.ntu.edu.sg, xiaofei.xfxie@gmail.com, mengguozhu@iie.ac.cn, yangliu@ntu.edu.sg

***Abstract*—Large-scale pre-trained models such as CodeBERT, GraphCodeBERT have earned widespread attention from both academia and industry. Attributed to the superior ability in code representation, they have been further applied in multiple downstream tasks such as clone detection, code search and code translation. However, it is also observed that these state-of-the-art pre-trained models are susceptible to adversarial attacks. The performance of these pre-trained models drops significantly with simple perturbations such as renaming variable names. This weakness may be inherited by their downstream models and thereby amplified at an unprecedented scale. To this end, we propose an approach namely ContraBERT that aims to improve the robustness of pre-trained models via contrastive learning. Specifically, we design nine kinds of simple and complex data augmentation operators on the programming language (PL) and natural language (NL) data to construct different variants. Furthermore, we continue to train the existing pre-trained models by masked language modeling (MLM) and contrastive pre-training task on the original samples with their augmented variants to enhance the robustness of the model. The extensive experiments demonstrate that ContraBERT can effectively improve the robustness of the existing pre-trained models. Further study also confirms that these robustness-enhanced models provide improvements as compared to original models over four popular downstream tasks.**

***Index Terms*—Code Pre-trained Models, Contrastive Learning, Model Robustness**

I. Introduction

It has already been confirmed that the "big code" era [1] is coming due to the ubiquitousness of software in modern society and the accelerated iteration of the software development cycle (design, implementation and maintenance). According to a GitHub official report [2] in 2018, GitHub has already reached 100 million hosted repositories. The Evans Data Corporation [3] also estimated that there are 23.9 million professional developers in 2019 and that number is expected to reach 28.7 million in 2024. As a result, the availability of code-related data is massive (e.g., billions of code, millions of changed code, bug fixes and code documentation), which yields a hot topic in both academia and industry. That is how to adopt the data-driven approach (e.g., deep learning) to solve conventional software engineering (SE) problems.

Deep learning has been widely applied to diverse SE tasks (AI4SE) such as software vulnerability detection [4], [5], [6], source code summarization [7], [8], [9], deep code search [10], [11] and source code completion [12], [13], [14]. Besides, the early works [15], [16], [17], [18], [19] directly utilized vanilla deep learning techniques such as Long-Short Memory Networks (LSTMs) [20] and Convolutional Neural Networks (CNNs) [21] for different tasks. Later works [5], [7], [10], [22], [23], [24], [4], [8] customized different network architectures to satisfy the characteristics of the specific task for achieving the best performance. For example, since complicated data dependencies and control dependencies are easier to trigger software vulnerabilities, Devign [5] incorporated different kinds of program structure information with Code Property Graph [25] to Graph Neural Networks [26] for vulnerability detection. Considering code duplication [27] is common in the "big code" era, Liu et al. [7] combined the retrieved code-summary pair to generate high-quality summaries. Although these customized networks have achieved significant improvements on specific tasks, the generalization performance is still low. To address this limitation, some researchers propose to utilize unsupervised techniques with the massive amount of data to pre-train a general model [28], [29], [30], [3], [31], [32], [33], [34], [35] and then fine-tune it for different downstream tasks. For example, CuBERT [36] pre-trained BERT [37] on a large collected Python corpus (7.4M files) and then fine-tuned it on different tasks such as variable-misuse identification and wrong binary operator identification. CodeBERT [29] pre-trained RoBERTa [38] for programming languages (PL) with their natural language (NL) comments on the open-source six programming languages [16] and evaluated it on code search and source code summarization. GraphCodeBERT [30] further incorporated data flow information to encode the relation of variables in a program for pre-training and demonstrated its effectiveness on four downstream tasks.

The aforementioned pre-trained models have a profound impact on the AI4SE community and have achieved promising results on various tasks. With the widespread use of pre-trained models, an important question is whether these models are robust to represent code semantics. Our preliminary study has demonstrated that state-of-the-art pre-trained models are not robust to a simple label-preserving program mutation such as variable renaming. Specifically, we utilize the test data of clone detection (POJ-104) [39] (a task to detect whether two functions are semantic equivalence with different implementations)

† Corresponding author.

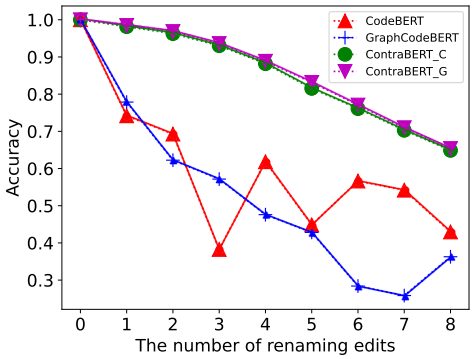


Fig. 1. Adversarial attacks on clone detection(POJ-104).

provided by CodeXGLUE [3] and select those samples that are predicted correctly by the pre-trained CodeBERT [29] and GraphCodeBERT [38]. Then we randomly rename variables within these programs from 1 to 8 edits. For example, 8 edits mean that we randomly select 8 different variables in a function and rename them for all occurrences with the newly generated names. If one function has less than 8 variables, we will rename the maximum number of variables. We then utilize these newly generated mutated variants to evaluate the model prediction accuracy based on the cosine similarity of the embedded vectors of these programs. Surprisingly, we find that either CodeBERT or GraphCodeBERT suffers greatly from renaming operation and the accuracy reduces to around 0.4 when renaming edits reach to 8 (see Fig. 1). It confirms that pre-trained models are not robust to adversarial examples. However, it is challenging to improve the robustness of pre-trained models. Although the latest work by Yang et al. [40] proposed some attack strategies to make CodeBERT and GraphCodeBERT have poor performance on adversarial samples. They further combined adversarial samples with original samples to fine-tune pre-trained models without any changes to the model architecture to improve prediction robustness on downstream tasks. However, a newly designed model that inherently solves the weakness of robustness is not involved in their paper.

In this paper, we propose ContraBERT, an unsupervised contrastive learning-based framework to enhance the robustness of existing pre-trained models in code scenarios. Compared with Yang et al. [40], we design a new pre-trained model that takes masked language modeling (MLM) and contrastive pre-training task as the pre-training tasks to improve model robustness. To design a contrastive pre-training task to help the model group similar samples while pushing away the dissimilar samples, we define nine kinds of simple or complex data augmentation operators that transform the original program and natural language sequence into different variants. Given an existing pre-trained model such as CodeBERT or GraphCodeBERT, we take the original sample as well as its augmented variants as the input to train the model with MLM and contrastive pre-training task, where MLM is utilized to help the model learn better token representations and contrastive pre-training task is utilized to help the model group the similar vector representations to enhance model robustness. As shown in Fig. 1, ContraBERT_C and ContraBERT_G denote the models are pre-trained from CodeBERT and GraphCodeBERT with our approach respectively, we observe that with the increasing number of edits, although the performance continues to drop, the curve for ContraBERT is much smoother. The prediction accuracy of ContraBERT_C and ContraBERT_G outperform CodeBERT and GraphCodeBERT significantly, indicating that ContraBERT_C and ContraBERT_G are more robust than the original models. We further perform an ablation study to confirm each type of defined PL-NL augmentation operator is effective to improve the model robustness. Finally, we conduct broad research on four downstream tasks (i.e., clone detection, defect-detection, code-to-code-trans and code search) to illustrate that these robustness-enhanced models provide significant improvements as compared to the original models. In summary, our main contributions are as follows:

- We present a framework ContraBERT that enhances the robustness of existing pre-trained models in the code scenario by the pre-training tasks of masked language modeling and contrastive learning on original samples as well as the augmented variants.
- We design nine kinds of simple or complex data augmentation operators on the programming language (PL) and natural language sequence (NL). Each operator confirms its effectiveness to improve the model's robustness.
- The broad research on four downstream tasks demonstrates that the robustness-enhanced models provide improvements as compared to the original models. Our code and model are released on [41] for reproduction.

Organization: The remainder of this paper is organized as follows: Section II describes the background of the original models that ContraBERT will use. We elaborate our approach in Section III. Section IV and Section V present the experimental setup and experimental results. In Section VI, we give some discussions about our work. After a brief review of related work in Section VII, we conclude this paper in Section VIII.

II. Background

In this section, we briefly introduce CodeBERT and GraphCodeBERT which will be adopted as our original pre-trained models for ContraBERT.

A. CodeBERT

CodeBERT [29] is pre-trained on an open-source benchmark CodeSearchNet [16], which includes 2.1M bimodal NL-PL (comment-function) pairs and 6.4M unimodal functions without comments across six programming languages. The model architecture is the same with RoBERTa [38], which utilizes multi-layer bidirectional Transformer [42] for unsupervised learning. Specifically, CodeBERT consists of 12 identical layers, 12 heads and the dimension size for each layer is 768. In total, the number of model parameters reaches 125M. Two different pre-training objectives are used, the first one is masked language modeling (MLM), which is trained on

bimodal data. MLM objective targets predicting the original tokens that are masked out in NL-PL pairs. To fully utilize unimodal data, CodeBERT further uses Replaced Token Detection (RTD) objective on both bimodal and unimodal samples. RTD objective is designed to determine whether a word is original or not. At the fine-tuning phase, two downstream tasks (i.e., code search and source code documentation generation) are used for evaluation. The experimental results demonstrate that CodeBERT outperforms supervised approaches on both tasks.

B. GraphCodeBERT

GraphCodeBERT [30] is a pre-trained model for code, which considers structures in code. Specifically, it incorporates the data flow of code to encode the relations of "where the value comes from" between variables in the pre-training stage. In addition to the pre-training task of masked language modeling (MLM), GraphCodeBERT further introduces two new structure-aware pre-training tasks. The first one edge prediction is designed to predict whether two nodes in the data flow are connected. The other node alignment is designed to align edges between code tokens and nodes. GraphCodeBERT utilizes NL-PL pairs for six programming languages from CodeSearchNet [16] for pre-training. It is fine-tuned on four downstream tasks including code search, clone detection, code translation and code refinement. The extensive experiments on these tasks confirm that code structures and the defined pre-training tasks help the model achieve state-of-the-art performance on these tasks.

III. Approach

In this section, we first present an overview of our approach, then detail each component including PL-NL augmentation, model design in pre-training and the fine-tuning settings for downstream tasks.

A. Overview

The overview of ContraBERT is shown in Fig. 2. Specifically, given a pair of the function C with its comment W (i.e., (C, W)), we first design a set of PL-NL augmentation operators $\{f(*)\}, \{g(*)\}$ to construct the simple or complex variants for C and W respectively. In the pre-training phase, initialized from existing pre-trained models such as CodeBERT or GraphCodeBERT, we further pre-train these models on the original samples and their augmented variants with masked language modeling (MLM) and contrastive pre-training task to enhance the model robustness. Finally, when ContraBERT is pre-trained over a large amount of unlabeled data, we fine-tune it for different types of tasks such as retrieval tasks, classification tasks and generation tasks with the task-specific data in a supervised manner.

B. PL-NL Augmentation

Given a program C, clone detection [43] could help to identify a semantically equivalent program C'. However, this technique is unrealistic in practice. For any function in a fixed dataset, we cannot guarantee that we will be able to find the

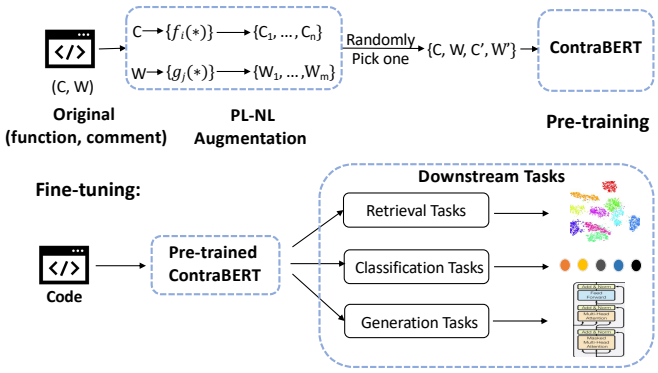


Fig. 2. The Overview of ContraBERT.

semantically equivalent variants. Furthermore, clone detection usually takes a project for analysis, which is not applicable to a single function. Hence, we consider constructing augmented variants based on the original samples. Compared with the existing works [44], [45] that only focus on program mutations, we design a set of natural language (NL) sequence augmented operators. Specifically, we design a series of simple operators and complex operators for both PL and NL to construct variants.

1) Program (PL) Augmentation Operators: For program augmented operators, we design four kinds of complex operators and one kind of simple operator.

Complex Operators:

- Rename Function Name (RFN). It is designed to replace the function name with a new name that is taken randomly from an extra vocabulary set constructed on the pre-training dataset. We extract all function names in the pre-training dataset for the construction. Since each sample in the dataset is a single function, the renamed function preserves the equivalent semantics to the original function.
- Rename Variable (RV). It renames variables in a function. A random number of variables for all occurrences in the function will be replaced with the new names taken randomly from an extra vocabulary set. We extract all variable names from the pre-training dataset to construct this vocabulary set. This operator only mutates the variable names and all occurrences of them with the new variable names, which does not change the semantics of the original function.
- Insert Dead Code (IDC). It means to insert unused statements in a function. To generate unused code statements, we traverse AST to identify the assignment statements and then randomly select one assignment statement to rename its variables with new names that have never appeared in the same function. After that, we consider it as the dead code and insert it at the position after the original assignment statement. As the inserted dead code does not change the original program behaviour, IDC is regarded as the semantically equivalent operator.
- Reorder (RO). It randomly swaps two lines of statements that have no dependency on each other in a basic block in a

function body such as two declaration statements appearing on two consecutive lines without other statements between them. We traverse AST and analyze the data dependency for extraction. Since the permuted statements are independent without data dependency, this operator preserves the original program semantics.

Simple Operators:

- Sampling (SP). It randomly deletes one line statement from a function body and preserves others. It can serve as regularizers to avoid overfitting [45].

2) Comment (NL) Augmentation Operators: Apart from the program augmentation, we further design one kind of complex operator and three kinds of simple operators for comment augmentation operators as follows:

Complex Operators:

- Back Translation Mutation (Trans). It refers to translating a source sequence into another language (target sequence) and then converting this target sequence to the original sequence [46]. We use the released tool [47] for the implementation where the source is in English and the target is in German.

Simple Operators:

- Delete. It randomly deletes a word in a comment.
- Switch. It randomly switches the positions of two words in a comment.
- Copy. It randomly copies a word and inserts it after this word in a comment.

Given a function C with its paired comment W, we utilize the above augmentation operators on C and W respectively to obtain the augmentation sets, which are defined as S_C and S_W respectively. Specifically, each operator is conducted once to get its corresponding augmented variant and insert it into the corresponding augmentation set. For the operator IDC, which may not get its variant for some specific functions, we ignore it and use other operators for the construction. Then we randomly select an augmented version from S_C and S_W (i.e., $C' \in S_C$ and $W' \in S_W$) and construct the quadruple (C, W, C', W') for the pre-training. Note that during the pre-training process, at each learning step, (C', W') is randomly selected from the augmented sets S_C and S_W respectively. Hence, each augmented sample in the sets is used when the model has sufficient learning steps.

C. Model Design and Pre-training

Basically, ContraBERT is further trained from existing pre-trained models. We directly utilize the existing pre-trained model and further pre-train it with masked language modeling (MLM) and contrastive pre-training task to enhance its robustness. The model design of ContraBERT is presented in Fig. 3.

1) Model Design: As shown in Fig. 3, ContraBERT consists of two separate encoders M and M', where M can be represented by any pre-trained models such as CodeBERT. The model architecture of M' is the same with the encoder M and the initial weights are also the same with M. However, the weight update strategy is different with M. Specifically, given a quadruple (C, W, C', W') from Section III-B, we construct two input sequences $X = \{[CLS], W, [SEP], C, [SEP]\}$ and $X' = \{[CLS], W', [SEP], C', [SEP]\}$, where "$[CLS]$" indicates the beginning of a sequence and "$[SEP]$" is a symbol that concatenates two kinds of sequence. We utilize the encoder M and M' to encode the masked input sequence X and X' respectively.

2) Pre-training Tasks: Masked language modeling (MLM) is an effective and widely adopted pre-training task to learn the effective token representations [37], [38], we also utilize it as one of our pre-training tasks. However, by our preliminary results in Section I, we observe that the models trained by MLM are weak to the adversarial examples, we further introduce a contrastive pre-training task to group the similar data and push away the dissimilar data to reshape the learnt space for encoder M to enhance the model robustness.

Masked Language Modeling (MLM). We utilize MLM to learn token representations in a sequence. Specifically, given the sequence $X = \{[CLS], W, [SEP], C, [SEP]\}$, a random set of positions in X are masked out. We select 15% tokens to mask out and obtain the masked token set. Furthermore, we replace 80% of the masked tokens in this set with the "[MASK]" symbol, 10% with the random tokens from the vocabulary set and the remaining 10% unchanged. We configure these settings since they are confirmed effective to learn the token representations in a sequence [37], [38]. The loss function $\mathcal{L}_{\text{MLM}}$ can be expressed as follows:

$$\mathcal{L}_{\text{MLM}} = -\sum_{x_i \in M} \log p(x_i | X^{mask}) \tag{1}$$

where X^{mask} is the masked input sequence and M is the masked token set.

Contrastive Pre-training. We design a contrastive pre-training task that uses InfoNCE [48] as the loss function to enhance model robustness. It can be expressed as follows:

$$\mathcal{L}_{\text{InfoNCE}} = -\log \frac{\exp(\boldsymbol{q} \cdot \boldsymbol{k}_+ / t)}{\exp(\boldsymbol{q} \cdot \boldsymbol{k}_+ / t) + \sum_{i=1}^{n} \exp(\boldsymbol{q} \cdot \boldsymbol{k}_i / t)} \tag{2}$$

where t is a temperature hyper-parameter [49], the query vector $\boldsymbol{q}$ is the encoded vector representation, $\boldsymbol{k}_+$ is a similar key vector that $\boldsymbol{q}$ matches, $\boldsymbol{K} = \{\boldsymbol{k}_1, ..., \boldsymbol{k}_n\}$ is a set of dissimilar encoded vectors. InfoNCE tries to classify the query vector $\boldsymbol{q}$ into its similar sample $\boldsymbol{k}_+$ and pushes it away from dissimilar samples in the set $\boldsymbol{K}$. The similarity is measured by dot product $(\cdot)$ between two vectors. To obtain the query representation $\boldsymbol{q}$ and the similar key representation $\boldsymbol{k}_+$, inspired by the recent advance [50] on the image recognition, we adopt Momentum Contrast (MoCo) [50] for the encoding. Specifically, it introduces an extra encoder M' to get the key representation $\boldsymbol{k}_+$, which can be expressed as follows:

$$\begin{aligned} \boldsymbol{q} &= \text{LayerNorm}(\text{M}(X)[0]) \\ \boldsymbol{k}_+ &= \text{LayerNorm}(\text{M}'(X')[0]) \end{aligned} \tag{3}$$

where X and X' denote the original masked sequence and its mutated variant respectively. The index 0 denotes the position

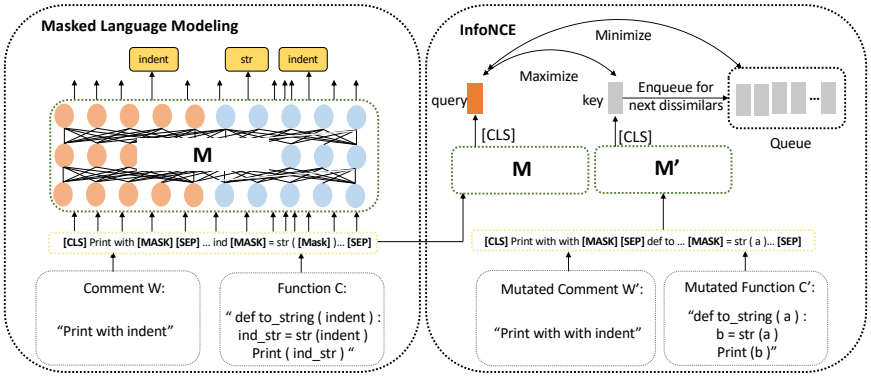


Fig. 3. The model design for ContraBERT where the encoder M can be represented by the existing pre-trained models such as CodeBERT. The initial weights of the encoder M' are the same as the encoder M while the weight update is different.

of "[CLS]" in the sequence, which can be considered as the aggregated sequence representation. The encoder M' is the same as the encoder M, but during the learning phase, it utilizes a momentum to update its learnt weights while the encoder M uses the gradient descent:

$$\theta_{\mathrm{M}'} \leftarrow m\theta_{\mathrm{M}'} + (1-m)\theta_{\mathrm{M}} \tag{4}$$

where $m \in [0, 1)$ is a momentum coefficient for scaling, $\theta_{\mathrm{M}'}$ and θ_{M} denote the learnt weights for model M' and M.

From Eq 3, we obtain the query representation $\boldsymbol{q}$ and the key representation $\boldsymbol{k}_+$, to compute the similarity with dissimilar vectors from $\boldsymbol{K}$, MoCo maintains a "*dynamic*" *queue* of length n. This queue stores the dissimilar keys from the previous batches. Specifically, during the learning phase, the current query $\boldsymbol{q}$ will calculate the similarity with all dissimilar vectors in this queue. Afterwards, the key vector $\boldsymbol{k}_+$ will be enqueued to *queue* to replace the oldest one and we take it as the dissimilar samples for the calculation of the next query. Hence, it is namely *dynamically updated*.

Finally, we add both loss values with the scaled factor to pre-train ContraBERT and this process is expressed as follows:

$$\mathcal{L}_{\mathrm{Loss}} = \mathcal{L}_{\mathrm{MLM}} + w\mathcal{L}_{\mathrm{InfoNCE}} \tag{5}$$

where w is the hyper-parameter to scale the weight for both pre-training tasks.

D. Fine-tuning

Once ContraBERT is further pre-trained from the original pre-trained model, we can utilize it to obtain the vector representation for a program. Furthermore, we can also transfer it to different downstream tasks during the fine-tuning phase. These downstream tasks can be roughly categorized into three groups: (1) retrieval tasks (e.g., clone detection [39], [51], code search [11], [16]); (2) classification tasks (e.g., defect-detection [5]); (3) generation tasks (e.g., code-to-code translation [52], [53], code-refinement [54] and source code summarization [7]). Since the output space may differ from the pre-trained space, similar to CodeBERT and GraphCodeBERT, we add the task-specific module and then fine-tune the completed network on the labeled data. Specifically, for retrieval tasks, we further train ContraBERT on a labeled dataset; for classification tasks, we add a multi-layer perceptron (MLP) to predict the probability for each class; for generation tasks, we add a Transformer-based decoder to generate the target sequence.

IV. Experimental Setup

In experiments, we first evaluate the effectiveness of our approach (RQ1) in improving model robustness. Then we plot the feature space learnt by different pre-trained models for visualization to confirm the features are learnt better(RQ2). Finally, we conduct extensive experiments to demonstrate the robustness-enhanced models provide significant improvements on downstream tasks (RQ3-RQ4). The detailed research questions are described as follows:

- **RQ1:** What is the performance of different augmentation operators in enhancing the robustness of the pre-trained model?
- **RQ2:** Can ContraBERT reshape the vector space learnt from the pre-trained models to obtain better vector representations?
- **RQ3:** Can ContraBERT outperform the original pre-trained models on different downstream tasks?
- **RQ4:** Are the defined pre-training tasks both effective in improving the downstream task performance?

A. Evaluation Tasks, Datasets and Baselines

We select four downstream tasks for evaluation. They are clone detection [39], [43], code search [16], defect detection [5] and code translation [52], [53]. We briefly introduce each task as follows:

Clone Detection (Code-Code Retrieval). This task is to identify semantically equivalent programs from a set of distractors by measuring the semantic similarity between two programs. AI for clone detection calculates cosine similarity between two embedding vectors of programs produced by neural networks and selects the top-k most similar programs as the candidates.

Defect Detection (Code Classification). It aims to detect whether a function contains defects that will be exploited to attack the software systems. Since the defects in a program are still difficult to be effectively detected by the traditional techniques, recently advanced works [5], [55], [17] propose to employ a deep neural network to learn program semantics to facilitate the detection. These AI-based techniques predict the probability of whether a function is vulnerable or not.

Code Translation (Code-Code Generation). It aims to translate a program in a programming language (e.g., Java) to the semantically equivalent one in another language (e.g., C#). Some previous works [53] analogy it to machine translation [42], [56] in NLP and employ LSTMs [20] and Transformer [42] for code translation.

Code Search (Text-Code Retrieval). It aims at returning the desired programs based on the query in a natural language. Similar to clone detection, it measures the semantic relevance between queries and programs. The input for the deep code search system [16], [11] is a natural language query and the output is programs that meet the query requirements. The cosine similarity is used to compute semantic similarity between the vectors of a query and programs.

In terms of the pre-training dataset, we use the released dataset provided by CodeSearchNet [16] and this dataset is also used by CodeBERT and GraphCodeBERT. We use bimodal NL-PL pairs for pre-training, which consist of six programming languages including Java, Python, Ruby, Go, PHP and JavaScript. For the fine-tuning datasets, for the tasks of clone detection (POJ-104), defect detection, and code translation, we directly utilize the released task-specific dataset provided by CodeXGLUE [3]. For code search, we use the cleaned dataset provided by GraphCodeBERT [30] for evaluation. For each task, we utilize the official scripts to make a fair comparison. In addition, by the defined augmentation operators in Section III-B, we obtain a large amount of extra data (C', W') used in ContraBERT as compared to the original pre-training data used in CodeBERT and GraphCodeBERT. Hence, we further add two baselines CodeBERT_Intr and GraphCodeBERT_Intr, which utilize original data as well as the dataset of the extra data (C', W') to pre-train CodeBERT and GraphCodeBERT with MLM for comparison.

B. Evaluation Metrics

In ContraBERT, different metrics are used to evaluate downstream tasks. We follow the metrics that CodeXGLUE used for evaluation, and the details are listed below:

MAP@R. It is the abbreviation of the mean of average precision, which is used to evaluate the result of retrieving R most similar samples in a set given a query. MAP@R is used for clone detection, where R is set to 499 for evaluation.

Acc. It defines the ratio of correct predictions (i.e., the exact match) in the testset. Acc is used for the evaluation of defect detection and code translation.

BLEU-4. It is widely used to evaluate the text similarity between the generated sequence with the ground-truth in the generation systems. We use BLEU-4 for code translation.

MRR. It is the abbreviation of Mean Reciprocal Rank, which is widely adopted in information retrieval systems [11], [57]. We used it to evaluate the performance of code search. Instead of retrieving 1,000 candidates like CodeBERT [29], we follow the settings of GraphCodeBERT [30] to retrieve the answer for each query from the whole test set.

C. Experimental Settings

We adopt CodeBERT and GraphCodeBERT as our original models. We set the maximum input sequence length X and the mutated sequence X' as 512 following CodeBERT. We use Adam for optimizing with 256 batch size and 2e-4 learning rate. At each iteration, X' is constructed by C' and W', which are randomly picked from S_C and S_W respectively. Following He et al. [50], the momentum coefficient m, temperature parameter t and *queue* size is set to 0.999, 0.07 and 65536 accordingly. We set the weight w in Eq 5 as 0.5 to accelerate the coverage process. The model is trained on a DGX machine with 4 NVIDIA Tesla V100 with 32GB memory. To alleviate the bias towards the high-resource languages (i.e., the number of samples for different programming languages is different), we refer to GraphCodeBERT [30] and sample each batch from the same programming language according to a multinomial distribution with probabilities $\{q_i\}_{i=1...N}$.

$$q_i = \frac{p_i^\alpha}{\sum_{j=1}^{N} p_j^\alpha} \; with \;\; p_i = \frac{n_i}{\sum_{k=1}^{N} n_k} \tag{6}$$

where n_i is the number of samples for i-th programming language, N is the total number of languages and α is set to 0.7. The model is trained with 50K steps to ensure each mutated sample is utilized for the learning process and it takes about 2 days to finish the pre-training process. At fine-tuning phase, we directly utilize the default settings of CodeXGLUE [3] and GraphCodeBERT [30] in ContraBERT for downstream tasks. All experiments of downstream tasks are conducted on Intel Xeon Silver 4214 Processor with 6 NVIDIA Quadro RTX 8000 with 48GB memory.

V. Experimental Results

A. RQ1: Robustness Enhancement.

We investigate the augmentation operators in enhancing model robustness by validating the accuracy of samples against adversarial attacks on clone detection (POJ-104). The main reason to choose clone detection is that it targets identifying the semantically equivalent samples from other distractors. Hence, although the variable renaming operator changes the text of a program, the original program semantics are still unchanged. We statistically analyse the correctly predicted results under a different number of renaming edits for illustration. The experiments are conducted in a zero-shot manner [58], which means that it does not involve fine-tuning phase and we directly utilize the pre-trained model for evaluation. Specifically, we remove one operator and keep the remaining operators in Section III-B to pre-train the model. For fairness, the other settings in the experiments are the same as ContraBERT. Then we utilize the testset (in total 12,000 samples) on clone

TABLE I. Results of ContraBERT against the variable renaming operator in a zero-shot manner.

Model	Num	N=0 Acc	N=1 Acc	N=4 Acc	N=8 Acc	Model	Num	N=0 Acc	N=1 Acc	N=4 Acc	N=8 Acc
ContraBERT_C w/o RFN	10,087	1	0.977	0.868	0.634	ContraBERT_G w/o RFN	10,375	1	0.975	0.873	0.634
ContraBERT_C w/o RV	8,665	1	0.932	0.597	0.291	ContraBERT_G w/o RV	9,042	1	0.955	0.657	0.309
ContraBERT_C w/o IDC	9,997	1	0.969	0.865	0.618	ContraBERT_G w/o IDC	10,530	1	0.963	0.862	0.612
ContraBERT_C w/o RO	9,923	1	0.963	0.857	0.619	ContraBERT_G w/o RO	10,509	1	0.968	0.868	0.617
ContraBERT_C w/o SP	10,604	1	0.959	0.849	0.616	ContraBERT_G w/o SP	11,140	1	0.969	0.860	0.613
ContraBERT_C w/o Trans	9,536	1	0.971	0.856	0.621	ContraBERT_G w/o Trans	10,360	1	0.973	0.859	0.617
ContraBERT_C w/o Delete	10,199	1	0.978	0.871	0.639	ContraBERT_G w/o Delete	10,376	1	0.981	0.878	0.643
ContraBERT_C w/o Switch	9,809	1	0.975	0.877	0.637	ContraBERT_G w/o Switch	10,457	1	0.978	0.876	0.647
ContraBERT_C w/o Copy	10,749	1	0.977	0.874	0.635	ContraBERT_G w/o Copy	10,859	1	0.981	0.880	0.641
ContraBERT_C	10,463	1	**0.981**	**0.882**	**0.649**	ContraBERT_G	10,565	1	**0.985**	**0.888**	**0.654**

detection (POJ-104) and randomly mutate the variables contained in the correctly predicted samples produced by different pre-trained models from 1 to 8 edits to test the prediction accuracy. The experimental results are shown in Table I where N is the number of edits and Num is the total number of correctly predicted samples without any edits in the testset for different models. ContraBERT_C/G defines the model is initialized by CodeBERT and GraphCodeBERT respectively and w/o $*$ defines the removed operator $*$.

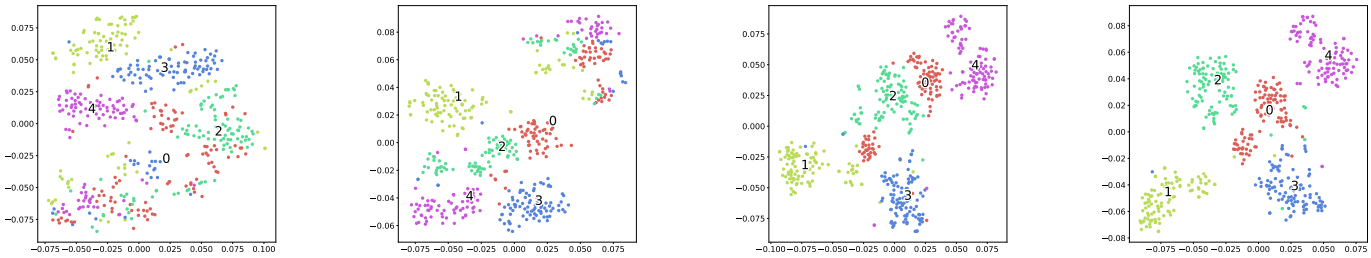
From Table I, we find that in general, with the increasing number of edits, the performance continues to drop. It is reasonable, as the increasing number of edits, the difficulty for corrected predictions is also increased. We also observe that each augmented operator is beneficial to improve model robustness against the adversarial samples and when incorporating all operators, we obtain the best performance. It demonstrates the effectiveness of our designed PL-NL augmentation operators. In terms of NL augmentation operators, the operators Delete/Switch/Copy are relatively weaker in the robustness enhancement compared with the operator Trans. Since the operators (Delete/Switch/Copy) just have a limited extent of modification on the original sequence (i.e., only one or two words are modified), the text similarity between W and W' is more similar than the operator Trans produces. Hence, the data diversity is limited by Delete/Switch/Copy, which leads to the robustness improvement is not as obvious as the operator Trans. In terms of PL augmentation operators, we find that the number of correctly predicted samples of ContraBERT_C/G w/o RV is the lowest (e.g., 8,665 and 9,042). With the increasing number of edits, the accuracy drops by a great margin. This indicates that RV operator plays a critical role against adversarial attacks and removing it harms the performance significantly. In addition, removing RFN operator, ContraBERT also has higher accuracy than other PL operators (i.e., RV, IDC, RO and SP), which indicates that RFN has fewer contributions. It is caused by the generated program C' by RFN (i.e., rename function name) is more similar to the original program C compared with other PL augmentation operators.

✍ ►**RQ1**◄ Each operator in the designed PL-NL augmentation is effective in improving model robustness and when incorporating them, the robustness of pre-trained models is further enhanced.

B. RQ2: Visualization for Code Embeddings.

We visualize the code representation space learnt by different pre-trained models to confirm that the contrastive pre-training task can reshape the learnt vector space to ensure the model is more robust. Specifically, we use the clone detection (POJ-104) task provided by CodeXGLUE [3] for evaluation. The main reason for selecting clone detection is that it is more intuitive to observe and validate the similarity of code representation on the semantic equivalence programs. The dataset consists of 104 programming problems, where each problem has 500 semantically equivalent programs with different implementations. Theoretically, the program semantics for one problem should be the same. Hence, the code vectors (i.e., representations) of programs from pre-train models for one problem should be closer than the code vectors of programs for other problems. We randomly select 5 different problems with 100 samples and take them as the inputs to CodeBERT, GraphCodeBERT, ContraBERT_C and ContraBERT_G for visualization where C/G defines ContraBERT is initialized by CodeBERT or GraphCodeBERT respectively. We utilize the vector of the token "[CLS]" as the program representation. We further utilize T-SNE [59] to reduce the vector dimension to a two-dimensional space for visualization. Similar to Section V-A, this process is also zero-shot [58], which helps us to validate the learnt space by different pre-training techniques.

As shown in Fig. 4, the vectors produced by GraphCodeBERT (See Fig. 4b) have a certain ability to group some problems of programs compared with CodeBERT (See Fig. 4a), which indicates that incorporating program structures such as data flow graph into pre-training is beneficial for the model to learn program semantics. However, we also find that the improvement is limited and the boundary in Fig. 4b is not clear. Some data points are scattered, especially in the upper-right part of Fig. 4b. In contrast, the visualization of ContraBERT is shown in Fig. 4c and Fig. 4d. We see that the programs in the same problem aggregate together closely as a cluster and different clusters have much clearer boundaries. This indicates that ContraBERT is more powerful than CodeBERT/GraphCodeBERT to group semantically equivalent data and push away dissimilar data. We attribute this ability to the defined PL-NL augmentation operators to capture the essence of programs. Furthermore, ContraBERT_G (See Fig. 4d) has a better clustering performance than ContraBERT_C (See Fig. 4c). For example in Fig. 4c, the label 0 has



(a) CodeBERT (b) GraphCodeBERT (c) ContraBERT_C (d) ContraBERT_G

Fig. 4. Visualization for vector representations of each 100 programs for 5 problems and they are randomly picked from clone detection (POJ-104). The vectors are produced by CodeBERT, GraphCodeBERT, ContraBERT_C and ContraBERT_G. The point with different colours indicates different problems that this function belongs to.

TABLE II. Results on clone detection and defect detection.

Model	Clone Detection MAP@R	Defect Detection Acc
CodeBERT	84.29	62.08
CodeBERT_Intr	86.34	62.41
ContraBERT_C (MLM)	86.21	62.25
ContraBERT_C (Contra)	81.44	62.22
ContraBERT_C	**90.46**	**64.17**
GraphCodeBERT	85.16	62.85
GraphCodeBERT_Intr	87.60	62.26
ContraBERT_G (MLM)	87.30	62.01
ContraBERT_G (Contra)	85.63	58.82
ContraBERT_G	90.06	63.32

TABLE III. Results on code translation.

Model	Code Translation			
	Java → C#		C# → Java	
	BLEU-4	Acc	BLEU-4	Acc
CodeBERT	79.92	59.00	72.14	58.00
CodeBERT_Intr	79.93	59.20	75.71	58.60
ContraBERT_C (MLM)	79.90	59.10	75.03	58.10
ContraBERT_C (Contra)	51.99	34.60	46.75	38.30
ContraBERT_C	79.95	59.00	75.92	59.60
GraphCodeBERT	80.58	59.40	72.64	58.80
GraphCodeBERT_Intr	80.61	59.60	75.50	60.10
ContraBERT_G (MLM)	80.36	59.40	75.10	60.00
ContraBERT_G (Contra)	55.48	39.40	48.92	39.00
ContraBERT_G	**80.78**	**59.90**	**76.24**	**60.50**

two distant clusters while in Fig. 4d, it only has one cluster. The improvements are from the used original model that GraphCodeBERT is superior to CodeBERT. In addition, we compute the distortion distance [1] [60] of the selected samples for these models to strengthen the conclusion. The distances of CodeBERT, GraphCodeBERT, ContraBERT_C and ContraBERT_G are 0.333, 0.212, 0.202, and 0.194 respectively. We can find that ContraBERT has a lower distortion distance than CodeBERT and GraphCodeBERT, which demonstrates their clusters are more compact.

✍ ►**RQ2**◄ Through contrastive pre-training tasks to learn augmented variants constructed by a set of PL-NL operators, ContraBERT is able to group the semantically equivalent samples and push away the dissimilar samples, thus learning better vector representations.

C. RQ3: Performance of ContraBERT on Downstream Tasks.

We conduct extensive experiments on four downstream tasks to evaluate the performance of ContraBERT as compared to the original CodeBERT and GraphCodeBERT. We further add two baselines (i.e., CodeBERT_Intr and GraphCodeBERT_Intr), which are pre-trained by original data as well as the augmented variants. We supplement these two baselines to ensure the used scale of data is consistent with ContraBERT for a fair comparison. The results of clone/defect detection are shown in Table II. Table III presents the results of code translation and Table IV presents the results of code search where the rightmost "overall" column is the average value for six programming languages. Because the values for clone detection and defect detection of GraphCodeBERT are not reported by their original paper [30], we use official code for reproduction and report these values. The other values of CodeBERT and GraphCodeBERT are directly taken from CodeXGLUE [3] and Guo et al. [30].

From Table II and Table III, we find that ContraBERT_C/G outperforms original pre-trained models CodeBERT or GraphCodeBERT on clone detection (POJ-104), defect detection and code translation. However, the absolute gains on code search (see Table IV) are minor. For these improvements, we attribute to the robustness-enhanced models providing better performance on downstream tasks. When it comes to minor improvements in code search, we ascribe to the difficulty of this task. Code search requires learning the semantic mapping between query and program. However, the semantic gap between programs and natural languages is huge. It makes the model difficult to achieve significant improvements. In total, considering the scale of testset on code search, which contains 52,561 samples for six programming languages, the improvements are still promising. Furthermore, we find that compared with CodeBERT and GraphCodeBERT, CodeBERT_Intr and GraphCodeBERT_Intr have better performance on these tasks. It is reasonable since we add extra data to further pre-train CodeBERT and GraphCodeBERT. However, the performance of CodeBERT_Intr and GraphCodeBERT_Intr is worse than

[1]The distortion distance refers to the sum of the squared distances of each sample to their assigned cluster centre.

TABLE IV. Results on code search where the evaluation metric is MRR.

Model	Ruby	Javascript	Go	Python	Java	PHP	Overall
CodeBERT	0.679	0.620	0.882	0.672	0.676	0.628	0.693
CodeBERT_Intr	0.686	0.623	0.883	0.676	0.678	0.630	0.696
ContraBERT_C (MLM)	0.675	0.621	0.888	0.670	0.675	0.631	0.693
ContraBERT_C (Contra)	0.593	0.532	0.864	0.622	0.618	0.584	0.636
ContraBERT_C	0.688	0.626	0.892	0.678	0.685	0.634	0.701
GraphCodeBERT	0.703	0.644	0.897	0.692	0.691	**0.649**	0.713
GraphCodeBERT_Intr	0.709	0.647	0.894	0.692	0.693	0.647	0.714
ContraBERT_G (MLM)	0.692	0.642	0.897	0.690	0.690	0.647	0.710
ContraBERT_G (Contra)	0.626	0.582	0.882	0.655	0.659	0.613	0.670
ContraBERT_G	**0.723**	**0.656**	**0.899**	**0.695**	**0.695**	0.648	**0.719**

ContraBERT_C/G. It demonstrates that even with the same scale of data, ContraBERT_C/G are still better than CodeBERT and GraphCodeBERT, which further strengthen our conclusion that the improvements are brought by our proposed approach rather than the gains brought by the increased scale of the data.

✍ ►**RQ3**◄ ContraBERT comprehensively improves the performance of original CodeBERT and GraphCodeBERT on four downstream tasks, we attribute the improvements to the enhanced robustness of the model has better performance on these tasks.

D. RQ4: Ablation Study for Pre-training Tasks.

ContraBERT utilizes two pre-training tasks, the first one is MLM, which learns the token representations and the second one is the contrastive pre-training task, which improves the model robustness by InfoNCE loss function. We further investigate the impact of each pre-training strategy on downstream tasks. The experimental results are shown in Table II, Table III and Table IV respectively, where the row of MLM or Contra denotes the results obtained by purely using MLM or contrastive pre-training task. For a fair comparison, the other settings are the same when combining both pre-training tasks for pre-training.

We can observe that the performance of purely using contrastive pre-training tasks is worse than purely using MLM on these downstream tasks, especially on the task of code translation. It is acceptable since both pre-training tasks are excellent in different aspects. Specifically, MLM is designed by randomly masking some tokens in a sequence to help the model learn token representations. The learnt token representations are important for generation tasks to generate a target sequence such as code translation, so it will help the model achieve good performance on these tasks. However, the contrastive pre-training task is designed by grouping the semantically equivalent samples while pushing away the dissimilar samples through InfoNCE loss function. The model robustness is enhanced by the contrastive pre-training task. Furthermore, when combing both pre-training tasks, our model achieves better performance compared with purely using one of the pre-training tasks, which indicates that ContraBERT is robust at the same time is able to achieve better performance on the downstream tasks.

✍ ►**RQ4**◄ Masked language modeling (MLM) and the contrastive pre-training task play different roles for ContraBERT. When combining them together, the model achieves higher performance on different downstream tasks.

VI. Discussion

In this section, we first discuss the implications of our work, then discuss the limitations followed by threats to validity.

A. Implications

In this work, we find that the widely used pre-trained code models such as CodeBERT [29] or GraphCodeBERT [30] are not robust to adversarial attacks. Based on this finding, we further propose a contrastive learning-based approach for improvement. We believe that this finding in our paper will inspire the following-up researchers when designing a new model architecture for code, considering some other problems in the model such as robustness, generalization and not just focusing on the accuracy of the model on different tasks.

B. Limitations

By our experimental results, we find that the robustness of the model is enhanced significantly compared with the original models. We attribute it to the contrastive pre-training task to learn the semantically equivalent samples. However, these robustness-enhanced models only have slight improvements on the downstream task of code search. For this task, since it requires learning the semantic mapping between a query and its corresponding code, the designed augmentation operators just modify the code or query itself, hence their correlations are not captured and this leads to the improvements being limited. For code search, a possible solution to further improve the performance is to build the token relations between PL and NL for augmented variants, however, it involves intensive work to analyse the relations between the program and natural language comment. We will explore it in our future work.

Another limitation is the designed augmentation operators for PL and NL. We just design some basic operators to transform programs and comments. These operators are straightforward, although they are confirmed their effectiveness in improving model robustness. It is intriguing to explore more complex augmentation strategies such as multiple operations on these operators for a sample to construct complex augmented variants.

C. Threats to Validity

Internal validity: The first threat is the hyper-parameter tuning for pre-training. More hyper-parameters need to tune than CodeBERT or GraphCodeBERT for example the temperature t, the momentum coefficient m and *queue* size. We follow the original settings from MoCo [50] and these parameters may not be optimal as they are designed for the task of image classification in computer vision. Due to that, the pre-training process is time-consuming and resource-consuming. We need nearly 2 days to complete one training process hence we ignore the hyper-parameter tuning process. However, we also find that even with the original parameters used in MoCo [50], ContraBERT still achieves higher performance than the original models. The second threat is that we use the same train-validation-test split that CodeXGLUE [3] and GraphCodeBERT [30] used. Adjusting the data split ratio or improving the training data quality may produce better results, however, we do not take these strategies to ensure a fair evaluation. The third threat is that we just use clone detection(POJ-104) to verify the robustness of the model is enhanced in Fig. 1 and Section V-A, we also plot the learnt space in Section V-B. The reason to select clone detection is that it aims at identifying the semantically equivalent programs from other distractors, which is suitable for the evaluation.

External validity: Some other pre-training works in the code scenario such as CuBERT [28] are not included for evaluation. CuBERT was pre-trained on a large Python corpus with MLM. Our approach is orthogonal to these pre-trained models and we just need to replace the encoder M with other existing pre-trained models for evaluation.

VII. Related Work

In this section, we briefly introduce the related works on contrastive learning, the pre-trained models for "big code" and the adversarial robustness of models of code.

A. Contrastive Learning

Contrastive learning is to learn representations by minimizing the distance between similar samples while maximizing the distance between different samples to help the similar samples closer to each other and different samples far apart from each other. Over the past few years, it has attracted increasing attention with many successful applications in computer vision [50], [61], [62], [63], [64], natural language processing [65], [66], [67], [68]. Recently, there are some works [44], [69], [45] that utilize contrastive learning for different software engineering tasks. For example, Bui et al. [44] proposed Corder, a contrastive learning approach for code-to-code retrieval, text-to-code retrieval and code-to-text summarization. VarCLR [69] aimed to learn the semantic representations of variable names based on contrastive learning for different downstream tasks such as variable similarity scoring and variable spelling error correction. ContraCode [45] generated variants by a source-to-source compiler on JavaScript and further combined these generated mutated samples with contrastive learning for the task of clone detection, type inference and code summarization. Compared with these existing works which only focus on designing mutated variants for code, we first illustrate the widely concerned CodeBERT and GraphCodeBERT are weak to the adversarial examples. Then we design a set of simple and complex augmented operators on both programs and natural language sequences to obtain different variants. By contrastive learning to learn semantically equivalent variants, the robustness of existing pre-trained models is enhanced. We further confirm that the robustness-enhanced models provide improvements on different downstream tasks.

B. Pre-trained Models for "Big Code"

Recently, pre-trained models are widely applied to the "big code" era [28], [29], [30], [3], [31], [32], [33], [34], [35], [45], [70]. For example, Kanade et al. [28] pre-trained CuBERT based on BERT [37] with a massive corpus of Python programs from GitHub and then fine-tuned it for some classification tasks such as variable misuse classification. Feng et al. [29] proposed CodeBERT, a bimodal pre-trained model for programming language (PL) and natural language (NL) that learns the program representation to support code search and source code summarization. GraphcodeBERT [30] combines the variable data-flow graph in a program with the code sequences and the natural language sequence to enhance CodeBERT. CodeXGLUE [3] also utilized CodeBERT and CodeGPT [71] to release a benchmark including several software engineering tasks. Liu et al. [70] proposed a CommitBART to support commit-related downstream tasks. Compared with existing pre-trained models, we illustrate they are not robust and further propose ContraBERT to enhance model robustness.

C. Adversarial Robustness on Models of Code

The research about adversarial robustness analysis on the models of code has attracted the attention [72], [73], [74], [40], [75], [76]. Generally, these works can be categorized into two groups: white-box and black-box manner, where the white-box means that the approach provides some explanations on the decision-making while the black-box mainly focuses on the statistical evaluation. In terms of white-box works, Yefet et al. [73] proposed DAMP to select the semantic preserving perturbations by deriving the output distribution of the model with the input. Srikant et al. [72] provided a general formulation of a perturbed program that models site locations and perturbation choices for each location. Then based on this formulation, they further proposed a set of first-order optimization algorithms for the solving. In terms of the black-box works, HMH [76] generated adversarial examples of the source code by conducting iterative identifier renaming and evaluated on source code functionality classification task. The latest work by Yang et al. [40] proposed ALERT to transform the inputs while preserving the optional semantics of original inputs by replacing the variables with the substitutes. Their

experiments are conducted on the pre-trained models CodeBERT and GraphCodeBERT. Compared with ALERT, which only designed the rename variable operation, in this paper, apart from the rename variable operation, we further design eight augmented operators over PL-NL pairs. Furthermore, a newly designed model to solve the weakness of robustness is not involved in ALERT. In contrast, we propose our general network architecture that uses contrastive learning to enhance model robustness. The extensive experiments have confirmed that our approach enhances the robustness of existing pre-trained models. We also demonstrate that these robustness-enhanced models provide improvements on different downstream tasks.

VIII. Conclusion

In this paper, we observe that state-of-the-art pre-trained models such as CodeBERT and GraphCodeBERT are not robust to adversarial attacks and a simple mutation operator (e.g., variable renaming) degrades their performance significantly. To address this problem, in this paper, we propose ContraBERT, a contrastive learning-based framework to enhance the robustness of existing pre-trained models by designing nine kinds of PL-NL augmented operators to group the semantically equivalent variants. Through extensive experiments, we have confirmed that the model's robustness is enhanced. Furthermore, we also illustrate that these robustness-enhanced models provide improvements on four downstream tasks.

IX. ACKNOWLEDGMENTS

We express our sincere gratitude to Mr Daya Guo from Sun Yat-sen University for his assistance. This research is partially supported by the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the Ministry of Education, Singapore under its Academic Research Tier 3 (MOET32020-0004). IIE authors are supported in part by NSFC (61902395), Beijing Nova Program. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[2] "The github blog," https://github.blog/2018-11-08-100m-repos/.

[3] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[4] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[5] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[6] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *arXiv preprint arXiv:2105.12787*, 2021.

[7] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," in *International Conference on Learning Representations*, 2020.

[8] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

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

[10] S. Liu, X. Xie, L. Ma, J. Siow, and Y. Liu, "Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search," *arXiv preprint arXiv:2111.02671*, 2021.

[11] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[12] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 329–340.

[13] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 245–256.

[14] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[15] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[16] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[17] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[18] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *arXiv preprint arXiv:1707.02275*, 2017.

[19] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[22] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, 2020.

[23] B. Wu, S. Liu, R. Feng, X. Xie, J. Siow, and S.-W. Lin, "Enhancing security patch identification by capturing structures in commits," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[24] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu, "Transrepair: Context-aware program repair for compilation errors," *arXiv preprint arXiv:2210.03986*, 2022.

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[26] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[27] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[28] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," 2019.

[29] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[30] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[31] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[32] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang *et al.*, "Exploring software naturalness through neural language models," *arXiv preprint arXiv:2006.12641*, 2020.

[33] R.-M. Karampatsis and C. Sutton, "Scelmo: Source code embeddings from language models," *arXiv preprint arXiv:2004.13214*, 2020.

[34] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[35] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[36] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[37] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[38] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[39] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[40] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," *arXiv preprint arXiv:2201.08698*, 2022.

[41] Authors, "Contrabert: Enhancing code pre-trained models via contrastive learning," https://sites.google.com/view/contrabert.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

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

[44] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 511–521.

[45] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

[46] R. Sennrich, B. Haddow, and A. Birch, "Improving neural machine translation models with monolingual data," *arXiv preprint arXiv:1511.06709*, 2015.

[47] E. Ma, "Nlp augmentation," https://github.com/makcedward/nlpaug, 2019.

[48] A. v. d. Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *arXiv preprint arXiv:1807.03748*, 2018.

[49] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin, "Unsupervised feature learning via non-parametric instance discrimination," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 3733–3742.

[50] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 9729–9738.

[51] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[52] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.

[53] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[54] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[55] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[56] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[57] J. Wang and J. Zhu, "Portfolio theory of information retrieval," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 2009, pp. 115–122.

[58] M. M. Palatucci, D. A. Pomerleau, G. E. Hinton, and T. Mitchell, "Zero-shot learning with semantic output codes," 2009.

[59] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[60] "K-means," https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[61] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International conference on machine learning*. PMLR, 2020, pp. 1597–1607.

[62] M. Kim, J. Tack, and S. J. Hwang, "Adversarial self-supervised contrastive learning," *arXiv preprint arXiv:2006.07589*, 2020.

[63] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," *arXiv preprint arXiv:2103.00020*, 2021.

[64] B. Dai and D. Lin, "Contrastive learning for image captioning," *arXiv preprint arXiv:1710.02534*, 2017.

[65] Z. Yang, Y. Cheng, Y. Liu, and M. Sun, "Reducing word omission errors in neural machine translation: A contrastive learning approach," 2019.

[66] H. Fang, S. Wang, M. Zhou, J. Ding, and P. Xie, "Cert: Contrastive self-supervised learning for language understanding," *arXiv preprint arXiv:2005.12766*, 2020.

[67] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," *arXiv preprint arXiv:2104.08821*, 2021.

[68] D. Shen, M. Zheng, Y. Shen, Y. Qu, and W. Chen, "A simple but tough-to-beat data augmentation approach for natural language understanding and generation," *arXiv preprint arXiv:2009.13818*, 2020.

[69] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," *arXiv preprint arXiv:2112.02650*, 2021.

[70] S. Liu, Y. Li, and Y. Liu, "Commitbart: A large pre-trained model for github commits," *arXiv preprint arXiv:2208.08100*, 2022.

[71] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[72] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U.-M. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," *arXiv preprint arXiv:2103.11882*, 2021.

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

[74] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," *arXiv preprint arXiv:2002.03043*, 2020.

[75] P. Bielik and M. Vechev, "Adversarial robustness for code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 896–907.

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