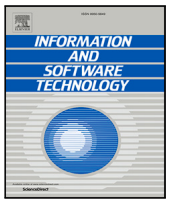
Information and Software Technology 181 (2025) 107699

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Assessing and improving syntactic adversarial robustness of pre-trained models for code translation

Guang Yang [a], Yu Zhou [a,*], Xiangyu Zhang [a], Xiang Chen [b], Tingting Han [c], Taolue Chen [c,*]

[a] Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b] School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China
[c] Birkbeck, University of London, United Kingdom

ARTICLE INFO

Keywords:
Code translation
Adversarial robustness
Pre-trained models
Data augmentation
Adversarial training

ABSTRACT

Context: Pre-trained models (PTMs) have demonstrated significant potential in automatic code translation. However, the vulnerability of these models in translation tasks, particularly in terms of syntax, has not been extensively investigated.
Objective: To fill this gap, our study aims to propose a novel approach CoTR to assess and improve the syntactic adversarial robustness of PTMs in code translation.
Methods: CoTR consists of two components: CoTR-A and CoTR-D. CoTR-A generates adversarial examples by transforming programs, while CoTR-D proposes a semantic distance-based sampling data augmentation method and adversarial training method to improve the model's robustness and generalization capabilities. The Pass@1 metric is used by CoTR to assess the performance of PTMs, which is more suitable for code translation tasks and offers a more precise evaluation in real-world scenarios.
Results: The effectiveness of CoTR is evaluated through experiments on real-world Java↔Python datasets. The results demonstrate that CoTR-A can significantly reduce the performance of existing PTMs, while CoTR-D effectively improves the robustness of PTMs.
Conclusion: Our study identifies the limitations of current PTMs, including large language models, in code translation tasks. It highlights the potential of CoTR as an effective solution to enhance the robustness of PTMs for code translation tasks.

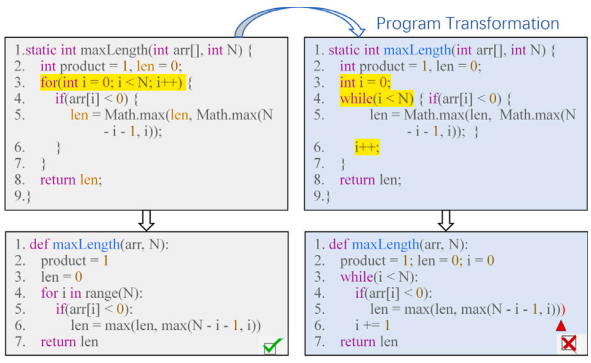
1. Introduction

Automated code translation is vital for seamless interoperability between systems and platforms during software migration [1,2]. It becomes particularly crucial when adopting new programming languages or modernizing legacy systems. However, manual code translation is time-consuming and error-prone [3]. For example, the migration of COBOL to Java at the Commonwealth Bank of Australia took about five years and $750 million to complete. To address this challenge, researchers have developed automated code translation tools, which have recently demonstrated great potential through the adoption of pre-trained models (PTMs) [4,5].

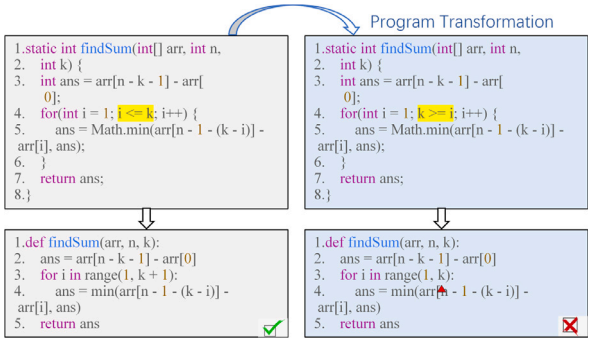
Despite the significant progress made in the field of PTMs, there are concerns regarding their accuracy and robustness in real-world scenarios. The previous studies [6–12] primarily focused on tasks, such as code summarization and method name prediction. In contrast, our study specifically focuses on the code translation task, which presents new challenges and needs dedicated research. During programming, there exist diverse alternatives to accomplish the same functionality. For example, one can use interchangeable for-loop or while-loop; or some developers may prefer `a<b` but others may prefer `b>a` when writing conditional statements. In the context of code translation, it is crucial to ensure these syntactically distinct yet semantically equivalent code snippets are translated into semantically equivalent target code. That means a *robust* (neural) code translation model should recognize the semantics while not overfitting the syntax of source code. However, in reality, even the most sophisticated tools (such as Copilot [13]) have faced criticism regarding their robustness [14,15]. For example, a minor syntactic difference can completely alter the behavior of a program, leading to serious bugs. Our goal is to improve the robustness of existing PTMs in code translation. This would enable software engineering researchers and practitioners to understand the strengths and weaknesses of PTM-based code translators. Furthermore, this would

* Corresponding authors.
E-mail addresses: yang.guang@nuaa.edu.cn (G. Yang), zhouyu@nuaa.edu.cn (Y. Zhou), zhangx1angyu@nuaa.edu.cn (X. Zhang), xchencs@ntu.edu.cn (X. Chen), t.han@bbk.ac.uk (T. Han), t.chen@bbk.ac.uk (T. Chen).

https://doi.org/10.1016/j.infsof.2025.107699
Received 26 October 2023; Received in revised form 11 February 2025; Accepted 14 February 2025
Available online 23 February 2025



(a) Syntactic errors in translation caused by loop exchange operation



(b) Functional errors in translation caused by condition exchange operation

Fig. 1. Syntactic and functional errors in translated code by CodeT5. In both examples, the original Java code (left) is transformed using different operations (highlighted in yellow), and then translated to Python by CodeT5 (right). (a) A while-loop is transformed into a for-loop, leading to syntactic errors in the translated code. (b) A condition statement is transformed by exchanging operands, resulting in functional errors in the translated output.

provide insights and guidelines for developing more robust models. To this end, we utilize adversarial attack techniques and generate adversarial examples, exposing vulnerabilities and weaknesses that may not be apparent in real-world scenarios or existing evaluation methods.

To illustrate the limitations of PTMs in code translation, we present two examples translated by CodeT5 [16] in Fig. 1 (from Java to Python).

In the initial example, the source code undergoes the "For/While Exchange" transformation (i.e., exchanging the while-loop for a for-loop, highlighted in yellow), leading to the detection of syntax errors in the CodeT5-translated code. In the second example, the source code undergoes 'Condition Exchange' transformation (highlighted in yellow), resulting in the emergence of functional faults in the CodeT5-translated code. These faults highlight the vulnerability of PTMs in handling subtle (syntactic) changes in code. Therefore, we provide a comprehensive investigation of the robustness of PTMs in code translation in this study.

In our study, we propose a novel approach CoTR (Code Translation Model Robustness Detector), comprising two essential components: CoTR-A and CoTR-D. CoTR-A imitates different programming styles through program transformation, attempting to generate code snippets to fail the model. In our study, this is referred to as an adversarial attack. Specifically, CoTR-A first defines a set of program transformation rules that are used to generate a collection of semantically equivalent source code. CoTR-A then feeds these code snippets into the victim model to identify the code that makes the model fail (i.e., does not pass all the test cases) as an adversarial code snippet. The low robustness of the model implies its sensitivity to the syntax of the input code, casting doubts about whether these seemingly well-performing models have truly learned essential code semantic features.

Different from AI security research, it is important to emphasize that the objective of CoTR-A is not to attack but to enhance the PTMs' performance. Consequently, CoTR-D adopts a dual-pronged strategy by retraining the victim model. Firstly, CoTR-D augments the training data using program transformation techniques. To mitigate the risk of overfitting, CoTR-D computes the semantic distance between the original data and the augmented data, selecting the sample with the maximum distance for sampling. Although this approach effectively enhances the model's robustness, it may lead to a reduction in accuracy for specific models. To tackle this concern, CoTR-D additionally adopts a gradient-based adversarial training method. Through this dual strategy, CoTR-D achieves noteworthy improvements in model robustness without compromising performance.

We conducted a large-scale empirical study involving 12 state-of-the-art PTMs on a real-world dataset. Our investigation reveals that existing PTMs can achieve the superior performance in code translation tasks. Regarding robustness, our study unfortunately indicates that the existing PTMs are not sufficiently robust when it comes to code translation. Specifically, our CoTR-A reduces the Pass@1 metric by at least 17.97% (from 17.97% to 43.02%) in the Java-to-Python dataset and by at least 14.29% (from 14.29% to 47.46%) in the Python-to-Java dataset. Furthermore, we observed that the existing pre-training techniques for model robustness (e.g., contrast learning [17] and adaptation learning [18]) are more adept at defending against token-based attacks [11,12] but are less sensitive to the syntactic transformations proposed in this study. Our findings also highlight the advantages of utilizing both data augmentation and adversarial training to enhance the robustness and generalization of code translation models.

We believe these findings are valuable for researchers and practitioners engaged in the field of code translation. For researchers, they provide valuable insights into the limitations and challenges faced by PTMs in code translation. This understanding can guide future research endeavors in developing more effective and reliable PTMs tailored specifically for code translation tasks. For practitioners, our study offers a practical solution to address the limitations of existing PTMs in code translation. By adopting our proposed tool, practitioners can enhance the accuracy of code translation, resulting in improved software quality.

The contributions of our study are summarized as follows.

- We construct high-quality datasets and comprehensively evaluate the functional accuracy and robustness of PTMs in code translation.
- We propose CoTR-A, which can effectively perform adversarial attacks on PTMs through program transformation.
- We propose CoTR-D, a defense method that can achieve significant improvements in model robustness without sacrificing its performance.

To facilitate the reproducibility of our study, we release source code, benchmarks, and experimental data at https://github.com/NTDXYG/COTR.

2. Preliminaries

2.1. Code translation

Code translation models take source code snippets as input and generate corresponding code snippets in the target language. In general, the model is trained on a labeled dataset $\mathcal{D}_{train} = (\mathcal{X}, \mathcal{Y}) := \{(x_1, y_1), \ldots, (x_N, y_N)\}$, where each $x_i \in \mathcal{X}$ (resp. $y_i \in \mathcal{Y}$) represents a source (resp. target) code snippet. Most pre-trained code translation models utilize the Transformer [19] architecture. The model $\mathcal{M}$, which comprises an encoder and a decoder, accepts the source code snippet $x \in \mathcal{X}$ as input and produces a sequence of hidden states $\mathcal{H}(x) = h_1(x), h_2(x), \ldots, h_n(x)$ as encoder's output. The decoder then accepts the

hidden states as well as the previously generated target code token $y_{1:t-1}$ as input to generate the probability distribution over the next target token y_t. This is achieved by passing the last decoder hidden state $\mathbf{s}_t$ through a linear layer followed by a softmax activation function

$$P_{\Theta_M}\left(y_t \mid y_{1:t-1}, x\right) = \text{softmax}\left(\mathbf{W}\mathbf{s}_t + \mathbf{b}\right),$$

where $\mathbf{W}$ and $\mathbf{b}$ are the learnable parameters of the linear layer. The negative log-likelihood is usually used as the loss function

$$\mathcal{L}\left(x, y; \Theta_M\right) = -\sum_{t=1}^{T} \log P_{\Theta_M}\left(y_t \mid y_{1:t-1}, x\right),$$

where T denotes the length of the target code sequence and Θ_M denotes the set of parameters of $\mathcal{M}$. During the training process, $\mathcal{M}$ is optimized to minimize the negative log-likelihood of the target code sequence presented given the source code sequence over the labeled data sampled from $\mathcal{D}_{train}$, i.e.,

$$\min_{\Theta_M} \mathbb{E}_{(x,y)\sim\mathcal{D}_{train}}\left[\mathcal{L}\left(x, y; \Theta_M\right)\right]$$

Please note that not all PTMs adopt the encoder–decoder structure. For instance, GPT-like PTMs solely consist of decoders, making the step where the encoder obtains hidden states optional.

2.2. Program transformation

Program transformation is a technique that modifies source code without compromising its overall functionality [20], and it has found extensive application in software engineering. The process of program transformation begins by parsing the code into an abstract syntax tree. Subsequently, depending on the transformation rule, the appropriate node is identified, and the transformation is executed accordingly. In general, program transformation can be formalized as a function $\mathcal{F}$ that takes the source code x and a set of transformation rules $\mathcal{R}$ as inputs and produces a set T of transformed code that satisfies the given constraints $\mathcal{G}$, i.e., $T = \mathcal{F}(x, \mathcal{R}, \mathcal{G})$.

To conduct a systematic review of the existing literature on program transformation, we employed a rigorous methodology. Firstly, we identified relevant keywords and conducted a comprehensive search for papers. We then manually screened the titles and abstracts of the papers to eliminate irrelevant ones. Additionally, we utilized academic search engines to supplement our search by checking citation status and exploring the list of published papers from relevant researchers. Finally, we have curated a list of program transformation rules from the literature (until March 2023), as presented in Table 1. These rules are categorized into three groups: 'Token Renaming', 'Statement Insert', and 'Statement Exchange'. These rules can be analyzed from three distinct aspects: semantics (S), informativeness (I), and readability (R). Semantics refers to whether the transformed code preserves the same functionality as the original code. Informativeness [21] pertains to whether the transformed code is consistent with the intended information expressed in the original code. Readability assesses whether the transformed code aligns with the human readability of the original code. It is important to note that in the literature, the concept of semantics can be understood from at least two different perspectives: one in the sense of formal semantics, capturing the functionality of the code, while the other is often referred to as "naturalness" [22], which treats the code as text in a natural language. In this study, we use semantics and informativeness to refer to these two perspectives of semantics, respectively.

Among these three types, we assert that only rules under the type of 'Statement Exchange' can maintain functional consistency, informativeness, and readability (refer to the 5th column of Table 1). On the other hand, the remaining two types of program transformation are highly likely to impact at least one of these three aspects. As an illustrative example, let us consider the Method Name Renaming rule. This rule involves modifying the method name in the code, but it may result in a loss of information. Specifically, the method name often contains valuable information about the functionality of the code from a natural language perspective. Replacing it with a generic name such as 'f' could lead to a loss of such essential information.

3. The CoTR approach

The framework of CoTR is illustrated in Fig. 2, which consists of two major components, CoTR-A (the upper part of Fig. 2) and CoTR-D (the lower part of Fig. 2).

3.1. Attack component: CoTR-A

To assess the robustness of the pre-trained model, we first fine-tune it on a given dataset $\mathcal{D}_{train}$, resulting in the creation of the victim model $\mathcal{M}$. This model maps each source code x to its corresponding target code $y = \mathcal{M}(x)$. Subsequently, we evaluate the performance of $\mathcal{M}$ on a designated test dataset $\mathcal{D}_{test} = \{(x_i, TC_i)\}$ to determine the accuracy of $\mathcal{M}$ in translating the source code correctly. To achieve this, we examine whether the translated target code $\mathcal{M}(x)$ for each x from $\mathcal{D}_{test}$ successfully passes all the provided test cases (TC) for x. This evaluation process is formulated as follows.

$$\boldsymbol{P}\left(\mathcal{M}, x_i, TC_i\right) = \begin{cases} 1, \text{ If } \mathcal{M}(x_i) \text{ passes all test cases } TC_i, \\ 0, \text{ otherwise.} \end{cases}$$

Intuitively, if the original output of $\mathcal{M}$ can successfully pass all the test cases, then the output code, even after experiencing minor perturbations to the input code, should also pass all the test cases. Therefore, in order to attack $\mathcal{M}$, we aim to generate an adversarial example x_{adv} for a given input x_i, which should be sufficiently similar to x_i but results in $\boldsymbol{P}\left(\mathcal{M}, x_{adv}, TC_i\right) = 0$.

Algorithm 1: Adversarial Attack Algorithm

Input: Fine-tuned Code Translation Model $\mathcal{M}$;
Code Translation DataSet with Test Cases $\mathcal{D}_{test}$;
Transformation Rules $\mathcal{R}$;
Transformation Constraint $\mathcal{G}$;
Output: Adversarial DataSet $\mathcal{D}_{adv}$;

```
Initialize Candidate Code Snippets T ← ∅;
Initialize Adversarial DataSet D_adv ← ∅;
for each (x, TC) ∈ D_test do
    if P(M, x, TC) == 0 then
        D_adv ← D_adv ∪ {x};
        break;
    T ← F(x, R, G)// Generate candidate code snippets
    if T is ∅ then
        D_adv ← D_adv ∪ {x};
        break;
    flag ← 0;
    for each x_t ∈ T do
        if P(M, x_t, TC_x) == 0 then
            D_adv ← D_adv ∪ {x_t};
            flag ← 1;
            break;
    if flag == 0 then
        D_adv ← D_adv ∪ {x};
return D_adv;
```

Algorithm 1 provides the pseudo-code of CoTR-A to describe the detailed attack process. The initial step of CoTR-A is to generate all possible adversarial code snippets $T = \mathcal{F}(x, \mathcal{R}, \mathcal{G})$ for each sample in $\mathcal{D}_{test}$ through program transformation. Subsequently, CoTR-A identifies the best code snippet for the original source code by minimizing the value of $\boldsymbol{P}$ to obtain the adversarial example. Formally, for each source code x from $\mathcal{D}_{test}$, we solve the following optimization problem

$$x_{adv} = \arg\min_{\hat{x}\in T} \boldsymbol{P}\left(\mathcal{M}, \hat{x}, TC_x\right)$$

Table 1
Comparison of program transformation rules, where S standards for semantics, I stands for informativeness and R stands for readability.

Type	Method	Description	Example	S	I	R
Token Renaming	API Renaming [23,24]	rename an API by other API names	`np.add()` → `np.sinc()`	×	×	×
	Arguments Renaming [7,10,23–26]	rename an augment by other words	`def f(size)` → `def f(a)`	✓	×	×
	Local Variable Renaming [7,8,10,23–33]	rename a local variable by other words and recursively update all related variables	`number=1` → `size=1`	✓	×	×
	Method Name Renaming [7,10,12,23–25,28,31,32]	rename a method by other words	`def count(a)` → `def f(a)`	✓	×	×
Statement Insert	Arguments Adding [23,24]	add an unused argument to function definition.	`def f(a)` → `def f(a, b)`	×	×	✓
	Dead Code Adding [8,23,24,26–28,30–32]	add an unreachable or unused code at a randomly selected location	add: `if (1==0): print(0)`	✓	×	×
	Duplication Code Adding [24,28,34]	duplicate a randomly selected assignment and insert it to its next line	`a=1;` → `a=1;a=1;`	✓	✓	×
	Filed Enhancement Adding [24]	enhance the rigor of the code by checking if the input of each argument is None	`def f(a):` → add: `if a is None: print(''ERROR'')`	✓	×	✓
	Plus Zero Adding [24,28]	select an numerical assignment of mathematical calculation and plus zero to its value	`a=1` → `a=1+0`	✓	✓	×
	Print Adding [23,24,26,30,33]	add a print line at a randomly selected location	add: `print(1)`	✓	×	×
	Return Optimal Adding [23,24]	change the return content to a variant with the same effect	`return 1` → `return 0 if (1==0) else 1`	✓	×	×
	TryCatch Adding [26,30]	add a single `try{A}catch(B){C}` statement	add: `try: catch():`	✓	×	✓
	UnrollWhiles Adding [26]	add a randomly selected, while loop in the target program has its loop body unrolled exactly one step	`while(A){B}` → `while(A){B;while(A){B} break;}`	✓	×	×
Statement Exchange	Loop Exchange [8,23,27,29–31,34]	replace a for loop with an equivalent while loop or replace a while loop with an equivalent for loop	`For` ⇔ `While`	✓	✓	✓
	Expression Exchange [27,29,34]	use the properties of expressions to transform	`a+=b` → `a=a+b`	✓	✓	✓
	Permute Exchange [8,27,30]	swap two independent statements in a basic block	`if(a){A} else{B}` → `if(!a){B} else{A}`	✓	✓	✓
	Condition Exchange [8,23,27,29–31]	reorder the left and right parts of a binary condition or transform `True` and `False` by logical operations	`if(a>b)` → `if(b<a)` or `True` → `!False`	✓	✓	✓
	Switch/If Exchange [8,30,34]	replace a switch statement with a if-else statement	`Switch` ⇔ `If/Else`	✓	✓	✓

To obtain x_{adv}, we define $\mathcal{D}_{adv} = \{x_{adv} \mid x \in \mathcal{D}_{test}\}$. If an adversarial example x_{adv} that successfully fools the model cannot be found, the original example x is straightforwardly added to $\mathcal{D}_{adv}$.

Step 1. Generation of Candidate Code Snippets. As mentioned in Section 2.2, for a given source code x, we construct its candidate code snippets using rule-based transformations. In the program analysis stage, we employ the third-party toolkit tree-sitter.[1] Regarding the transformation rules, we initially establish two constraints, denoted as $\mathcal{G}$: (1) The variant code should maintain functional consistency, ensuring it passes all test cases as the original code does. (2) The variant code should also be consistent with the original code in terms of informativeness and readability.

As presented in Table 1, considering informativeness and readability, we exclusively generate candidates for the 'Statement Exchange' category of transformation rules. It is worth noting that not all programming languages support 'switch' statements (e.g., Python only introduced 'match' statements as an alternative to 'switch' statements in v3.10). Therefore, we consider the following four rules from the last category, 'Statement Exchange', namely:

- **Rule-L**: Loop Exchange;
- **Rule-E**: Expression Exchange;
- **Rule-P**: Permutation Exchange;
- **Rule-C**: Condition Exchange.

These rules distinguish themselves from mutation operators in mutation testing as they are designed to maintain the semantic, readability, and informativeness consistency between the variant code and the original code. Detailed functional descriptions and code examples for these rules are available in Table 1.

To implement these transformation rules, we use tree-sitter to parse the source code and extract the abstract syntax tree (AST) out of the code. We then apply the transformation rules to AST to generate the

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

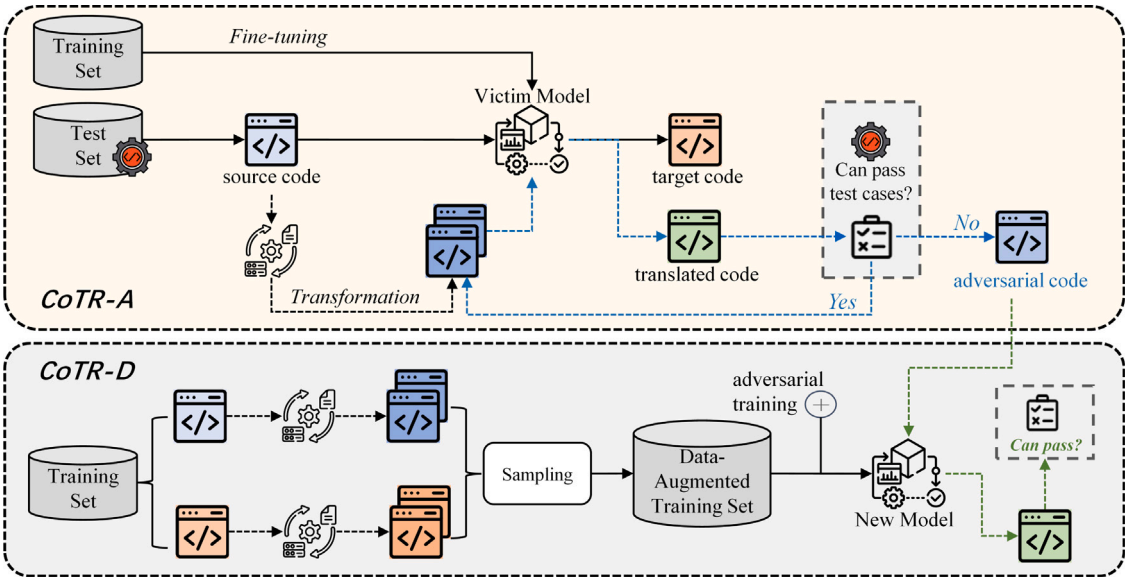


Fig. 2. The framework of CoTR.

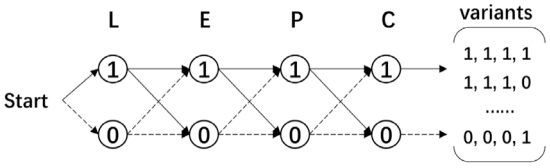


Fig. 3. Illustration of Candidate Code Snippet Generation.

candidate code snippets. For **Rule-L**, considering the characteristics of programming languages, we transform `for` loops to `while` loops in Java code and vice versa. For Python code, we transform `for` loops to `while` loops. For **Rule-E**, we implement the transformation of the five operators `+=`, `-=`, `*=`, `/=` and `%=`. For example, we rewrite `a += b` to `a = a + b`. For **Rule-P**, we first extract the `if-else` statement blocks from the code and then swap the order of `if` and `else`. We also change the conditional expressions in the `if` statement. We implement the transformation of logical operators `&&`, `&`, `||`, `|`, and relational operators `<`, `>`, `<=`, `>=`, `==`, `!=`. For **Rule-C**, we first extract the expressions from the code and then transform the expressions that satisfy the condition `a operator b` to `b operator a`. For this rule, we only consider the transformation of one operator between two different variables.

Note that these transformation rules are not mutually exclusive; after applying one rule, others can still be utilized to transform the source code. To improve the diversity of candidate code snippets, we take into account transformation sequences over **L**, **E**, **P**, **C**. To ensure the diversity of candidate code snippets while minimizing the search space, we generate at most one code snippet for each rule. In instances where multiple locations in the code can be transformed (e.g., multiple occurrences of the `+=` operator), we randomly select one for transformation. An illustration is provided in Fig. 3, where the four rules serve as input parameters, each with a value of 0 or 1, indicating whether the rule is used for transformation or not. We exhaustively enumerate all strings over $\{L, E, P, C\}$, which gives a search space where each string denotes a transformation sequence. By applying these transformations, we can generate adversarial attacks.

It is essential to emphasize that the use of exhaustive heuristics aims to ensure the discovery of the most challenging adversarial examples. While heuristic algorithms can reduce search costs to some extent [11], their results may be influenced by prior assumptions and heuristic rules. Consequently, they may generate adversarial samples that are not optimal or most challenging. In contrast, the exhaustive approach traverses all possible input variants, guaranteeing that no potential adversarial examples are overlooked. Although the computational complexity of this method is higher, it provides a more reliable assurance that the generated adversarial examples possess a high attack success rate.

Step 2. Selection of Adversarial Example. This step is designed to identify the most effective adversarial examples within the search space, which can successfully deceive the victim model. As adversarial samples are typically generated from inputs that can be accurately processed by the victim model [32], we exclude inputs that the victim model cannot process correctly. For a given dataset *Dtest*, we initially verify whether the code translated by the victim model $\mathcal{M}$ can pass all the test cases. If it fails to do so, we add this code to D_{adv} (Lines 4–6).

Next, CoTR-A generates all variant code snippets as candidates by exhaustively considering all possible combinations of the four transformation rules and verifying whether the generated candidates are empty. If the candidate set turns out to be empty, we include the original example in D_{adv} (Lines 7–10). As we adopt a rule-based approach, not all code will be successfully transformed. Therefore, the time cost of using the search-based approach is deemed acceptable. For the generated candidates, we traverse through them to identify the adversarial example that can effectively attack the victim model (Lines 11–16). Finally, in the event that no candidate can successfully attack the victim model, we add the original example to D_{adv} (Lines 17–18).

3.2. *Defense component:* COTR-D

As mentioned previously, it is crucial for adversarial code to retain functionality while maintaining the same level of informativeness and readability. In order to augment the training dataset, we require source-target code snippet pairs where the program transformation rule is applied to both the source and target code. To ensure the quality of the augmented data samples, constructing test cases for each sample becomes necessary. However, this process can be time-consuming, laborious, and even error-prone. Additionally, adding all variants to the augmentation dataset significantly increases the risk of model overfitting.

Data Augmentation. In light of this, we adopt a distinct data augmentation strategy in CoTR-D. Specifically, we employ a semantic

distance-based sampling method which can construct the augmented dataset more efficiently. First, for each source-target code pair (x, y) in the training set, we apply the transformation rules to generate a variant pair (x', y'). Then, we leverage the capabilities of CodeBERT [35] as a semantic feature extractor to calculate the semantic distance between the original code and its variant. This enables us to select the most diverse variants for augmentation. This process can be formalized as

$$\mathcal{D}_{aug} \sim \max_{\substack{x' \in F(x,\mathcal{R},\mathcal{G}) \\ y' \in F(y,\mathcal{R},\mathcal{G})}} \text{Distance}\left[f(x, x', y, y')\right]$$

where $f(x, x', y, y') = CodeBERT(x, x') + CodeBERT(y, y')$.

We proceed to dataset augmentation $\mathcal{D}_{aug}$ by selecting the variant code snippet with the largest semantic distance from the $\mathcal{D}_{train}$. We calculate this distance by computing the cosine distance between the original source code and its variants, and also between the original target code and its variants. The sum of the two distance values is finally used. By adopting this approach, we effectively enhance the diversity of the training set while mitigating the risk of overfitting.

Adversarial Training. It is observed in our empirical study (cf. Section 5.2) that data augmentation techniques have the potential to reduce the model's accuracy [10,12,36]. To address this issue, we adopt a noisy-enhanced adversarial training method N-PGD based on Projected Gradient Descent [37].

In general, the PGD algorithm operates by iteratively perturbing the input data x in the direction that maximizes the loss function, while ensuring that the perturbations remain within a specified epsilon bound. This iterative process is repeated for a fixed number of iterations, and the model parameters are updated during training based on the results. The general principle of PGD can be summarized by

$$\min_{\Theta_M} \mathbb{E}_{(x,y)\sim D_{gra}} \left[\max_{\Delta x \in \Omega} \mathcal{L}\left(x + \Delta x, y; \Theta_M\right) \right]$$

where Δx represents the perturbation applied to x, which is computed by the learning rate and the norm gradient of the x. Ω denotes the specified epsilon bound. The set $\mathcal{D}_{gra}$ refers to the gradient-based augmented dataset.

4. Experiments

To evaluate the effectiveness and benefits of our proposed approach, we mainly investigate the following two research questions (RQs):

RQ1: How robust are existing pre-trained models under CoTR-A?

In this RQ, we investigate the performance of existing pre-trained models on code translation tasks to discuss the feasibility of applying these models to code translation tasks. We then evaluate the robustness of existing pre-trained models using CoTR-A and other token-based attack methods to study the robustness of these models to different perturbations. By comparing the impact of different attack methods on existing pre-trained models, we can evaluate the effectiveness of CoTR-A. Finally, we conduct a human study to discuss the differences of adversarial examples generated by CoTR-A and other token-based attack methods.

RQ2: How effective is CoTR-D in improving the robustness of existing PTMs for code translation?

In this RQ, we evaluate the effectiveness of CoTR-D in improving the robustness of existing pre-trained models for code translation tasks. Meanwhile, we evaluate the impact of different components in CoTR-D on improving the robustness of existing pre-trained models.

4.1. Datasets

To assess the effectiveness of CoTR, we employ AVATAR, a compilation of the Java/Python dataset obtained from competitive programming sites, online platforms, and open-source repositories [38]. In order to construct clean and high-quality datasets, as well as to facilitate the creation of test cases, we design four heuristic rules:

Table 2
Length statistics of samples in the corpus.

Language	Avg.	Mode.	Median.	<128	<256
Java	100	79	90	72.5%	100%
Python	95	62	86	76.0%	100%

H1 Extract function-level code and perform syntax compilation check.

H2 Remove code with input tokens such as 'input()', 'args *', etc.

H3 Remove duplicate code.

H4 Remove code with inconsistent method names for better readability.

After applying these heuristic rules, we obtain a set of 3000 pairs of data samples, consisting of 2600 pairs in the training set, 200 pairs in the validation set, and 200 pairs in the test set. To ensure the quality of the test cases, we employ 10 postgraduate students, each has 3–5 years of programming experience. Each student is tasked to construct test cases for the samples in the test set, and each sample is evaluated using five test cases. To enhance the coverage of test cases, we implement a cross-checking process, wherein each student writes test cases for 20 code snippets and verifies their work with others. Additionally, students are permitted to search for relevant information and unfamiliar concepts on the Internet. To prevent fatigue and maintain accuracy, we impose a limit on each student to write a maximum of 50 test cases within a half-day. Table 2 presents the statistical details of the Java and Python code in our dataset.

4.2. Baseline attack methods

In this study, we propose a novel adversarial attack method, CoTR-A, a syntactic transformation-based attack method that satisfies the semantics, informativeness and readability. To evaluate the effectiveness of CoTR-A, we compare it with two token-based attack methods, i.e., RADAR [12] and ALERT [11].

RADAR considers semantic equivalence, typos and visual similarity, as simple typos are known to be significant in code refactoring. ALERT utilizes CodeBERT and GraphCodeBERT to generate natural candidates and employs a combination of greedy search and genetic algorithm for optimization.

It is worth mentioning that CoTR-A can be compatible with existing token-based methods and can be combined with them to provide a more comprehensive evaluation of PTMs' robustness. Therefore, we consider the attack method named "Combine", which combines CoTR-A with these two token-based attack methods to generate adversarial examples for evaluating the performance impact of PTMs as the number of rules increases.

4.3. Evaluation metrics

In this study, we consider different performance metrics to evaluate the code translation models, including

- **BLEU** [39], which is widely used to measure the similarity between the translated and the reference code based on the n-gram precision.
- **Code-BLEU** [40], an extension of BLEU which considers keywords, syntax, and data flow of the translated code.
- **EM**, which measures the percentage of cases where the translated code exactly matches the reference code.
- **Code-Exec** [41], which examines the syntax of code to guarantee that there are no syntax errors, type errors, or other errors that could hinder the execution of the code.

Table 3
Hyperparameters and their value.

Hyperparameter	Value
Optimizer	AdamW (BAdamW)
Random Seed	1234
Learning Rate	5e−5
Batch size	16
Beam size	10
Max input length	350
Max output length	350

- **P_s@1 (Pass@1)** [42], which is the percentage of the translated code that passes the test cases, i.e., the code which is deemed to be functionally correct.

For the robustness of the model, we consider two specific metrics.

- **RP_s@1 (Robust $Pass_s$@1)** [31], which is the percentage of the translated code that passes the test cases after the adversarial attack.
- **RD_s@1 (Robust $Drop_s$@1)** [31], which means the relative performance change between P_s@1 and RP_s@1, defined as

$$RD_s@1 = 1 - \frac{\text{Robust Pass}_s@1}{\text{Pass@1}}$$

4.4. Victim pre-trained models

We select ten widely used pre-trained models specialized for code translation tasks. These models can be classified into three groups based on their architecture: Encoder-only (Enc), Decoder-only (Dec), and Encoder-Decoder (Enc-Dec) models. In addition, we consider large models with billions of parameters which are classified as large language models (LLMs).

- **Enc:** In our study, we consider three representative Encoder-only models, including CodeBERT [35], GraphCodeBERT [43], and ContraBERT [17].
- Dec: We consider three representative Decoder-only models, including CodeGPT [18], CodeGPT-adapter [18], and CodeGen [44].
- Enc-Dec: We consider four representative Encoder-Decoder models, including NatGen [27], CodeT5 [16], PLBART [45], and UniXcoder [46].
- LLM: We also consider the two widely-used large language models, CodeLlama [47] and DeepSeek-Coder [48].

All pre-trained models and corresponding tokenizers are loaded from the official repository Huggingface.[2] To ensure a fair comparison, we maintain consistent hyper-parameters for all models throughout our study. In light of the computational resources, we opted for the BAdam optimizer [49] for two LLMs to fine tune. The BAdam optimizer is a variant of the Adam optimizer that can reduce the memory consumption of the model. The hyper-parameters and their respective values are summarized in Table 3.

Our implementation is based on PyTorch 1.8, and the experiments are conducted on a machine with an Intel(R) Xeon(R) Silver 4210 CPU and the GeForce RTX 3090 GPU.

5. Results

5.1. RQ1: How robust are existing pre-trained models under COTR-A?

5.1.1. Performance comparison

Table 4 presents the results, including evaluation metrics and model parameters, for comparative analysis. The best result is highlighted in bold, and the second-best result is underlined. Our findings indicate that not all PTMs can effectively translate high-quality code. Specifically, NatGen and CodeT5 demonstrate significantly better performance compared to other models, achieving a pass@1 metric of more than 70; in contrast, CodeBERT and ContraBERT exhibit inferior performance, with pass@1 metrics of less than 40. Meanwhile, LLMs like CodeLlama and DeepSeek-Coder achieve the best performance, with pass@1 metrics of 80.50 and 79.50, respectively. This result is consistent with the findings of Chen et al. [42], who reported that LLMs outperform other PTMs in code translation tasks.

Evaluation metrics. Our investigation reveals that the existing automatic evaluation metrics may not faithfully assess the functional correctness of translated code. For instance, the BLEU and CodeBLEU metrics of NatGen are higher than those of CodeT5, but the Pass@1 metric of its performance is lower. This phenomenon is illustrated in Fig. 1, where translated code may resemble the reference code but fail compilation or some test cases.

5.1.2. Robustness comparison

In our empirical study (Table 5), we present metrics (such as RPs@1 and RDs@1). Higher RPs@1 values or lower RDs@1 values indicate greater model robustness. The best results are highlighted in **boldface**.

Robustness degradation. We evaluate model robustness using the RDs@1 metric, where a higher RDs@1 value indicates lower robustness. For example, the CodeT5 model achieves a Ps@1 value of 76% when translating Java to Python (cf. Table 4). However, under the CoTR-A attack, RPs@1 decreases to 60%, and the RD_s@1 increases to 21.05%, indicating a significant 21.05% performance reduction. Comparing Tables 4 and 5, we observe performance degradation across all PTMs, with RDs@1 values ranging from 14.29% to 47.46%. We conclude that these models are generally *not* robust for code translation. Among the different models, LLMs like CodeLlama and DeepSeek-Coder exhibit the best robustness performance (their RDs@1 value can also be maintained at around 11.32% to 16.13% under CoTR-A's attack). Conversely, the Encoder-only models show the least robustness.

Attack effectiveness. Table 5 shows that CoTR-A generally outperforms RADAR and ALERT in terms of attack effectiveness, except for PLBART and GraphCodeBERT. The vulnerability of the Encoder-only model to token-based attacks is noteworthy, likely due to the random initialization of its decoder parameters and the lack of pre-training. Additionally, Shi et al. [50] suggest that lower model layers tend to concentrate on lexical properties, while higher layers focus on syntactic and semantic properties. This finding may explain the superior performance of syntax-based attacks compared to token-based attacks. Furthermore, it is worth highlighting that combined attacks 'Combine' can have a considerably greater impact on the performance of PTMs. By leveraging the strengths of both CoTR-A and token-based attack algorithms, the combined approach poses a more significant challenge.

Pre-training techniques. We also evaluate the impact of different pre-training techniques on model robustness. NatGen incorporates a de-naturalizing pre-training task, focusing on the naturalness of code, which leads to performance that outperforms CodeT5. ContraBERT incorporates contrast learning, leading to better RD@1 results but worse RP_s@1 results compared to GraphCodeBERT. CodeGPT-adapter utilizes adapter learning and demonstrates improved performance over CodeGPT. From Table 5, we observe that these pre-training techniques may be effective in defending against token-based attacks but are less effective against syntax-based attacks like CoTR-A.

5.1.3. Human study

To evaluate the quality of adversarial code, we further conduct a human evaluation study. We collect code snippets that can be attacked by RADAR, ALERT, and CoTR-A in the above experiments, resulting in a total of 159 pairs. We invite five graduated students who have 3~5 years of experience in Java and Python to participate in the

[2] https://huggingface.co/models

Table 4
Comparison results between different PTMs.

Type	Model	Parameters	Java-to-Python					Python-to-Java				
			BLEU	*Code-BLEU*	*EM*	*Code-Exec*	P_s@1	*BLEU*	*Code-BLEU*	*EM*	*Code-Exec*	P_s@1
Enc-Dec	NatGen	223M	84.36	80.57	27.00	97.50	73.50	82.82	82.08	18.00	84.50	71.00
	CodeT5	223M	83.30	79.52	27.00	97.50	76.00	83.11	81.81	13.00	84.00	70.00
	PLBART	139M	83.14	79.07	23.50	89.00	70.00	60.38	67.69	6.00	37.00	22.50
	UniXcoder	127M	81.63	78.58	24.00	90.50	64.00	81.38	80.59	9.50	74.50	58.00
Enc	CodeBERT	173M	75.12	71.99	10.50	66.00	40.50	76.03	74.87	5.00	45.00	29.50
	GraphCodeBERT	173M	76.33	73.63	12.00	73.50	43.00	77.45	75.70	10.00	50.00	36.50
	ContraBERT	173M	75.47	72.70	9.50	72.50	38.00	75.87	74.35	9.00	38.50	29.50
Dec	CodeGPT	124M	80.89	76.72	19.50	85.00	57.50	76.91	76.04	17.00	67.00	49.50
	CodeGPT-adapter	124M	82.18	78.20	27.50	92.00	67.00	79.07	77.98	16.50	72.50	57.00
	CodeGen	355M	81.35	78.09	17.00	90.50	59.50	79.50	79.03	15.00	68.50	51.50
LLM	CodeLlama	7B	**88.24**	**85.13**	**33.00**	**99.00**	**80.50**	**85.74**	**84.97**	**25.00**	**90.00**	**78.50**
	DeepSeek-Coder	6.7B	87.12	83.98	31.50	98.50	79.50	84.50	83.87	24.50	89.50	77.50

Table 5
Comparison results between different attack methods.

Model	Attack	Java-to-Python		Python-to-Java		Model	Attack	Java-to-Python		Python-to-Java	
		RP_s@1	RD_s@1	RP_s@1	RD_s@1			RP_s@1	RD_s@1	RP_s@1	RD_s@1
NatGen	RADAR	70.50	4.08	68.50	3.52	CodeBERT	RADAR	28.00	30.86	18.50	37.29
	ALERT	68.50	6.80	67.00	5.63		ALERT	29.50	27.16	22.00	25.42
	CoTR-A	59.50	19.05	60.00	15.49		CoTR-A	24.50	39.51	14.50	47.46
	Combine	**57.00**	**22.45**	**59.00**	**16.90**		Combine	**21.00**	**48.15**	**12.00**	**59.32**
CodeT5	RADAR	71.00	6.58	62.50	9.29	GraphCodeBERT	RADAR	22.00	48.84	24.50	32.88
	ALERT	68.50	9.87	61.50	12.14		ALERT	27.50	36.05	26.00	28.77
	CoTR-A	60.00	21.05	60.00	14.29		CoTR-A	24.50	43.02	26.00	28.77
	Combine	**55.50**	**26.97**	**56.00**	**20.00**		Combine	**15.00**	**65.12**	**17.00**	**53.42**
PLBART	RADAR	56.00	20.00	11.50	48.89	ContraBERT	RADAR	24.00	36.84	20.50	30.44
	ALERT	55.50	20.71	12.00	46.67		ALERT	30.50	19.74	19.50	33.90
	CoTR-A	47.00	32.86	17.00	24.44		CoTR-A	22.50	40.79	18.50	37.29
	Combine	**42.00**	**40.00**	**11.00**	**51.11**		Combine	**18.50**	**51.32**	**13.50**	**54.24**
UniXcoder	RADAR	56.00	12.50	46.50	19.83	CodeGPT	RADAR	44.50	22.61	39.50	20.20
	ALERT	56.50	11.72	49.00	15.52		ALERT	44.50	22.61	38.00	23.23
	CoTR-A	52.50	17.97	40.50	30.17		CoTR-A	36.00	37.39	36.50	26.26
	Combine	**46.50**	**27.34**	**34.50**	**40.52**		Combine	**30.50**	**46.96**	**33.00**	**33.33**
CodeLlama	RADAR	75.00	6.83	71.00	9.55	CodeGPT-adapter	RADAR	62.50	6.72	51.50	9.65
	ALERT	75.00	6.83	72.00	8.28		ALERT	60.50	9.70	52.50	7.89
	CoTR-A	71.00	11.80	68.00	13.38		CoTR-A	45.50	32.09	40.00	29.82
	Combine	**67.00**	**16.77**	**65.50**	**16.56**		Combine	**40.50**	**39.55**	**34.00**	**40.35**
DeepSeek-Coder	RADAR	75.00	5.66	70.50	9.03	CodeGen	RADAR	54.50	8.40	47.50	7.77
	ALERT	72.00	9.43	71.00	8.39		ALERT	55.50	6.72	48.50	5.83
	CoTR-A	70.50	11.32	65.00	16.13		CoTR-A	45.00	24.37	37.50	27.18
	Combine	**65.00**	**18.24**	**62.00**	**20.00**		Combine	**41.50**	**30.25**	**32.00**	**37.86**

Table 6
Results of our human study.

Approach	Informativeness	Readability
RADAR	3.25	3.47
ALERT	3.45	3.50
CoTR-A	**3.64**	**3.55**

evaluation.

To conduct the evaluation, we generate a questionnaire (shown in Fig. 4) for each code snippet and ask each participant to score the informativeness and readability of three adversarial examples generated by RADAR, ALERT, and CoTR-A. The scores range from 0 to 4, with higher scores indicating better quality. To ensure a fair comparison, the source of the adversarial code is hidden from the participants, and the order of the questionnaires is randomized. The workload of each participant is restricted, not exceeding 50 code snippets in half a day, to ensure the quality of evaluation.

The results of the human evaluation study are presented in Table 6, which shows the average scores given by the participants for each adversarial code snippet in terms of informativeness and readability. We observe that CoTR-A outperforms RADAR and ALERT on both aspects, with an improvement of 0.39 and 0.08 respectively. This suggests that CoTR-A generates adversarial code that is more informative and readable than those generated by RADAR and ALERT, further verifying the superiority of our approach.

Summary of RQ1

(1) The empirical study reveals that the existing PTMs are generally not robust for code translation tasks. (2) The syntactic transformation-based attack method CoTR-A can outperform token-based attacking methods on most models.

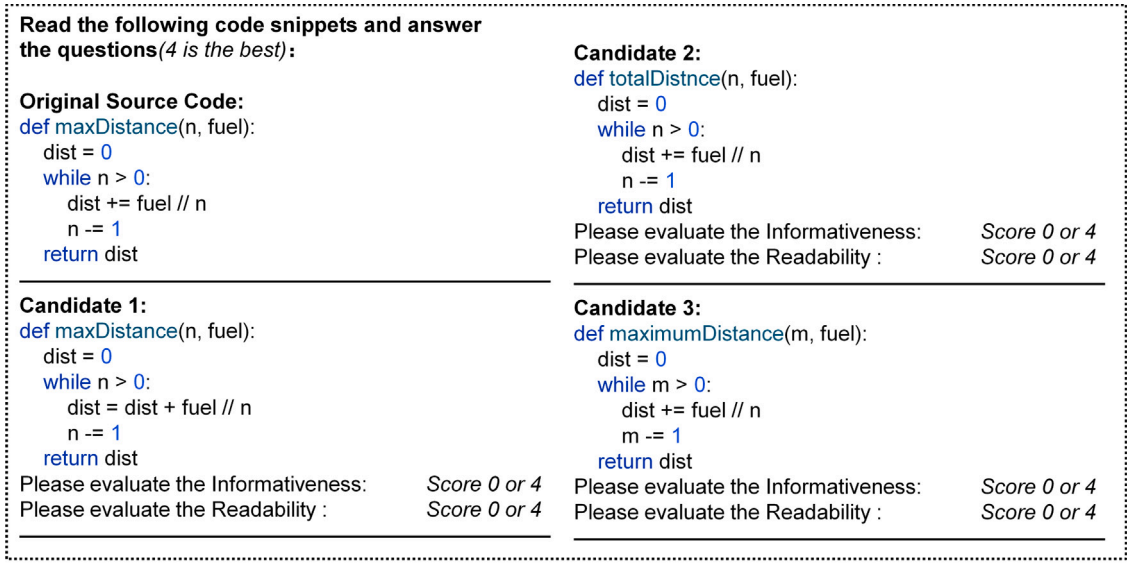


Fig. 4. Sample questionnaire used in the human evaluation.

Table 7
Comparison results between different defense methods in the Java-to-Python dataset.

Type	Model	P_s@1				RP_s@1				RD_s@1			
		Original	DA	AT	CoTR-D	Original	DA	AT	CoTR-D	Original	DA	AT	CoTR-D
Enc-Dec	NatGen	73.50	74.00	**80.50**	79.50	59.50	70.50	69.50	**75.00**	19.50	**4.73**	14.29	5.66
	CodeT5	76.00	69.50	**77.50**	74.50	60.00	69.00	66.50	**74.00**	21.05	0.72	14.19	**0.67**
	PLBART	**70.00**	51.50	69.50	68.00	47.00	50.00	56.00	**65.50**	32.86	**2.91**	19.42	3.68
	UniXocder	64.00	69.00	**71.50**	69.00	52.50	67.50	66.00	**68.50**	17.97	2.17	7.69	**0.72**
Enc	CodeBERT	40.50	43.00	**47.50**	43.50	24.50	39.00	**40.00**	**40.00**	39.51	9.30	15.79	**8.75**
	GraphCodeBERT	43.00	41.50	**48.50**	44.50	24.50	36.50	39.00	**40.50**	43.02	12.05	19.59	**8.99**
	ContraBERT	38.00	42.50	46.00	**48.50**	22.50	40.00	35.00	**42.00**	40.79	**5.88**	23.91	15.48
Dec	CodeGPT	57.50	58.00	53.50	**61.50**	36.00	53.50	41.50	**55.50**	37.39	**7.76**	22.43	9.76
	CodeGPT-adapter	67.00	63.50	61.50	**67.50**	45.50	61.00	46.00	**64.00**	32.09	**3.94**	25.20	5.19
	CodeGen	59.50	**67.00**	61.00	**67.00**	45.00	**66.00**	57.00	**66.00**	24.37	**1.49**	6.56	**1.49**
LLM	CodeLlama	80.50	82.00	83.50	**85.50**	71.00	78.50	75.00	**84.50**	11.80	5.49	10.18	**1.17**
	DeepSeek-Coder	79.50	83.50	83.00	**86.00**	70.50	82.50	81.00	**84.00**	11.32	**1.20**	2.41	2.33

Table 8
Comparison results between different defense methods in the Python-to-Java dataset.

Type	Model	P_s@1				RP_s@1				RD_s@1			
		Original	DA	AT	CoTR-D	Original	DA	AT	CoTR-D	Original	DA	AT	CoTR-D
Enc-Dec	NatGen	71.00	**73.00**	72.00	**73.00**	60.00	66.50	**68.00**	67.50	15.49	8.90	**5.56**	7.53
	CodeT5	70.00	61.50	**72.50**	71.00	60.00	58.00	**67.00**	66.50	14.29	**5.69**	7.59	6.34
	PLBART	22.50	44.00	16.50	**56.50**	17.00	41.50	15.50	**55.00**	24.44	5.68	6.06	**2.65**
	UniXocder	58.00	63.50	55.00	**66.50**	40.50	56.50	47.50	**58.00**	30.17	**11.02**	13.64	12.78
Enc	CodeBERT	29.50	32.00	32.00	**36.00**	15.50	31.00	27.00	**33.00**	47.46	**3.13**	15.63	8.33
	GraphCodeBERT	36.50	**43.50**	33.00	41.00	26.00	39.50	29.00	**40.00**	28.77	9.20	12.12	**2.44**
	ContraBERT	29.50	**38.50**	36.00	36.50	18.50	**38.50**	33.50	36.00	37.29	**0.00**	6.94	1.37
Dec	CodeGPT	49.50	47.50	49.50	**50.00**	36.50	43.00	44.50	**45.00**	26.26	**9.47**	10.10	10.00
	CodeGPT-adapter	**57.00**	51.50	55.00	**57.00**	40.00	47.50	50.50	**51.00**	29.82	**7.77**	8.18	10.53
	CodeGen	51.50	54.00	56.50	**60.50**	37.50	50.50	49.00	**57.00**	27.18	6.48	13.27	**5.79**
LLM	CodeLlama	78.50	80.00	81.00	**82.00**	68.00	75.50	78.00	**79.50**	13.38	5.63	3.70	**3.05**
	DeepSeek-Coder	77.50	79.00	80.00	**81.50**	65.00	76.00	79.00	**80.00**	16.13	3.80	**1.25**	1.84

5.2. RQ2: How effective is CoTR-D *in improving the robustness of existing PTMs for code translation?*

To assess the impact of CoTR-D on enhancing the robustness of PTMs for code translation, we conducted a comparison for different models in terms of the Pass@1, RPs@1, and RDs@1 metrics. These models include the original model without any defense mechanism, the model with data augmentation, the model with adversarial training, and the model with our proposed CoTR-D method. The experimental results for the Python to Java translation task can be found in Table 7, while the results for the Java to Python translation task are presented in Table 8.

(a) Dimensional distribution on the Java dataset

(b) Dimensional distribution on the Python dataset

Fig. 5. Dimensional distribution of original and augmentation data.

Data augmentation (DA). DA has demonstrated effectiveness in enhancing model robustness; however, it may not be adequate to guarantee optimal performance. As observed in the results, DA can provide an advantage in terms of the RDs@1 metric, indicating improved robustness. However, there could be a potential degradation in the Ps@1 metric, signifying reduced performance. This phenomenon can be attributed to certain models excessively emphasizing the augmented data during the fine-tuning process, which can be understood from the perspective of data distribution.

DA is effective in improving model robustness, but it may not be sufficient to ensure its performance. As seen in the results, DA is able to show an advantage in the RD_s@1 metric, but there may be a performance degradation in the P_s@1 metric. The reason is that some of the models are overly about the augmented data part in the fine-tuning process, which can be explained from the perspective of data distribution. Indeed we analyze the distribution relationship between the original and augmented data by visualizing the semantic feature representations using CodeBERT. We apply PCA [51] to obtain graphs of the different datasets. We map each sample into a 512-dimension vector through CodeBERT and the mean-pooling operation, and then project the vector into a two-dimensional plane using PCA, as shown in Fig. 5.

The results of the data distribution analysis are shown in Fig. 5. From the distributions of the original and augmented datasets, We can see that they are very similar. The slight difference lies in, for instance, for the distribution of the python dataset, the original data is skew toward the upper part of the semantic space, while the enhanced data is skew toward the lower part. This indicates that the data enhancement method successfully maintains the original data distribution and expands it accordingly.

Adversarial training (AT). AT has demonstrated effectiveness in enhancing model performance; however, it is often less robust than data augmentation (DA) in terms of improving model robustness. As observed in the results, AT can provide an advantage in the Ps@1 metric, indicating improved performance. However, there could be a potential degradation in the RDs@1 metric, signifying reduced robustness. This finding suggests that gradient-based AT is useful in improving the model's performance, but it may not be sufficient on its own.

Our proposed CoTR-D approach combines the strengths of both DA and AT techniques to improve the robustness and performance of pre-trained models (PTMs) in code translation tasks. By leveraging DA to generate diverse training data and AT to train models on adversarial examples, CoTR-D ensures that the model's performance remains stable while significantly enhancing its robustness against various types of attacks. For instance, the RDs@1 values for most of the models are kept within 10%, while their Ps@1 and RP_s@1 values are better than the original models.

Table 9
Assessing the Robustness of LLMs for Code Translation.

Task	Model	Pass@1	RP_s@1	RD_s@1
Java-to-Python	GPT-3.5-turbo	87.50	80.50	8.00
	GPT-4o	90.50	83.00	8.29
Python-to-Java	GPT-3.5-turbo	79.50	56.00	29.56
	GPT-4o	85.50	70.00	18.13

Summary of RQ2

Employing data augmentation or adversarial training techniques alone may damage the model's performance or robustness. However, our proposed CoTR-D approach effectively combines these two techniques, resulting in improved model robustness without sacrificing the translation accuracy.

6. Discussion

6.1. Robustness of larger models

Although we have incorporated two open-source large models in our experiments, the validation of CoTR-D on larger models is limited by hardware resource constraints and the closed-source nature of some models. However, we aim to explore the robustness of CoTR-A by validating it on larger models.

We note that in recent years, the size of PTMs is exploding. As the size of language models and training data continue to increase, larger models demonstrate various emergent abilities [52]. One such ability is zero-shot learning, which allows models to answer within a specific instruction or prompt [53]. In particular, LLMs have achieved excellent performance and demonstrated promising potential on code translation tasks [42]. To further verify the robustness of LLMs and the effectiveness of our attack method, we discuss the zero-shot performance of GPT-3.5-turbo.[3] and GPT-4o[4]

The final results can be found in Table 9. We find that GPT's performance is strong in zero-shot scenarios and outperforms both CodeLlama and DeepSeek-Coder with supervised learning, but the robustness issue still exists for both GPT-3.5-turbo and GPT-4o. The results show that the robustness of LLMs is still a challenge in code translation tasks. We believe that our proposed CoTR-A can be applied to larger models to enhance their robustness in code translation tasks.

6.2. Threats to validity

Threats to internal validity. Firstly, we mitigate implementation errors by conducting thorough checks on our implementation and utilizing mature libraries. Additionally, we have ensured the functionality of the variant code generated by CoTR-A by test cases. Secondly, to ensure a comprehensive evaluation of different model types, we have chosen ten state-of-the-art models by covering three diverse types of models.

While our experiments included local versions of CodeLlama and DeepSeek-Coder, they represent smaller variants of currently available models. The rapid advancement in language models means that significantly larger models, such as Llama3 (70B, 405B) [54] and DeepSeek R1 [55], could potentially yield different results. Furthermore, many state-of-the-art LLMs (including GPT-3.5-turbo and GPT-4) are closed-source commercial models, making fine-tuning experiments impossible. This limitation is particularly noticeable given the trend towards using these proprietary models in industry.

[3] https://platform.openai.com/docs/models/#gpt-3-5-turbo
[4] https://platform.openai.com/docs/models/#gpt-4o

Future research directions include evaluating our approach on these larger, more resource-intensive models to provide a more comprehensive understanding of robustness across different model scales.

Threats to external validity. Our dataset is derived from code competitions, and thus may not fully reflect the complexity of real-world scenarios. However, it provides valuable initial insights into the challenges of robust code translation. Importantly, our approach is language-independent, and the proposed enhancements can be applicable to different programming languages.

Additionally, our study is limited to method-level code translation, which, according to the survey [56], is consistent with most existing datasets in literature. However, it excludes important object-oriented programming concepts such as inheritance hierarchies, external library interactions and encapsulation patterns. This constraint may impact our findings' generalizability, particularly when considering more complex software architectures and systems.

In future research, we plan to expand our study to include more diverse and complex programs to validate the effectiveness of the proposed enhancements on a larger scale.

Threats to construct validity. Performance measure selection is the main construct threat. To mitigate this, we selected five widely used performance measures to evaluate the translation quality of our models. Additionally, to assess the robustness of our models against adversarial attacks, we introduce two specific metrics, RPs@1 and RDs@1, which focus on the success rate and diversity of the attacks, respectively. Furthermore, we conducted a human study to analyze the quality of our generated adversarial code.

7. Related work

7.1. Code translation

Early studies utilized rule templates or statistical methods to perform translations between different programming languages. For instance, phrase-based models were employed to translate code from C# to Java or from Python2 to Python3 [57,58]. In a later study, An et al. [59] proposed a rule-based approach that inferred syntactic transformation rules and API mappings to automatically translate Java code to Swift. Zhong et al. [60] explored the use of Application Programming Interfaces (APIs) in the context of code translation. However, these approaches are typically limited to a few specific language pairs and often require the creation of parallel datasets either manually or through rule-based tools.

In recent years, attention has shifted towards neural network based approaches (in particular, pre-trained models) for code translation. Roziere et al. [3] proposed TransCoder, an unsupervised pre-trained model based on unsupervised machine translation. Roziere et al. [61] showed that augmenting TransCoder with de-obfuscated targets can significantly improve performance. Liu et al. [5] proposed SDA-Trans, a syntax and domain-aware model for program translation. Meanwhile, supervised approaches have also proven successful, and the ten code pre-training models mentioned in this paper have all achieved impressive results when used for code translation as a downstream task after fine-tuning.

7.2. Adversarial attack and defense on code-related models

The robustness of neural network models has been extensively studied, particularly in image classification tasks. However, there is also a growing body of research focusing on code-related tasks, such as source code classification (Code↦Label) [34], code summarization (Code↦NL) [10], and code generation task (NL↦Code) [12,31].

Adversarial attacks on code can manifest in two forms: token-based attacks and syntax-based attacks. Token-based attacks predominantly focus on code identifiers and manipulate the model by replacing tokens with equivalent semantics. For instance, Zhang et al. [7] proposed MHM, which utilizes Metropolis–Hastings sampling-based identifier renaming. Zeng et al. [9] employed a wide range of NLP-based adversarial attack methods to evaluate pre-trained models and discovered that random attack methods can outperform carefully designed adversarial attack methods in most cases. Recent research has increasingly emphasized addressing the naturalness aspect of adversarial examples. Yang et al. [11] proposed a naturalness-aware attack called ALERT, which generates multiple natural candidates using GraphCodeBERT and CodeBERT. Zhou et al. [10] proposed ACCENT, which generates multiple natural candidates using the word2vec. Zhang et al. [32] introduced CARROT, an optimization-based attack technique that assesses and improves the robustness of deep program processing models. Yang et al. [12] proposed RADAR, which generates semantic and visual similar adversarial examples for code generation. Jha and Reddy [25] proposed CodeAttack, which finds the most vulnerable tokens and then substitutes these vulnerable tokens to generate adversarial examples.

Syntax-based attacks are primarily concerned with the syntax of the code and manipulate the model through transformations that preserve syntactic equivalence. Pour et al. [23] introduced a search-based testing framework for deep neural networks of source code embedding. Their framework focused on "for-loop enhance" and "if-loop enhance" to target code syntax. They applied this framework to tasks such as method name prediction, code captioning, code search, and code documentation generation. Rabin et al. [8] conducted an evaluation of multiple syntactic transformations on code search, code summarization, and code analogies. However, their study did not consider combinations of these transformations.

Adversarial defense on code models can be categorized as either active or passive. Active defense approaches involve re-training models with adversarial examples to enhance their robustness. For instance, Zhang et al. [7] proposed adversarial training as an active defense method for code translation tasks. In contrast, passive defense approaches aim to restore model performance without re-training or modifying the model. Zhou et al. [10] introduced a lightweight adversarial training method called the mask training algorithm. Yang et al. [12] also proposed a passive defense approach for code generation tasks through method name generation.

In contrast to previous work, we focus on program transformation based attacks instead of token-based attacks. Additionally, we investigate the impact of combining different program transformation methods, providing insights into the factors that contribute to the non-robustness of existing pre-trained models. Furthermore, we explore and employ a variety of defensive approaches to enhance model robustness and generalization in the face of adversarial attacks. Our study aims to contribute to a comprehensive understanding of the vulnerabilities and defenses in the context of code translation tasks.

8. Conclusion

In this study, we have conducted a thorough investigation of the robustness of pre-trained models in code translation tasks. We present CoTR, a novel approach that aims to assess and enhance the robustness of these models. Our research exposes the limitations of existing PTMs, including larger models such as GPT-3.5-turbo and GPT-4o, in effectively handling code translation tasks. To address these limitations, we propose CoTR-D, a defense mechanism that demonstrates promising results in improving the robustness and generalization of PTMs. Our findings provide valuable insights into the challenges and potential solutions for building more robust code translation models.

In future work, we plan to develop a more robust pre-trained model that can handle different programming styles and syntax conventions. We also plan to explore the use of other techniques, such as program repair or LLMs, to improve the effectiveness and robustness of pre-trained models in handling code translation tasks. Finally, we aim to construct a more comprehensive dataset that includes more diverse and complex programs to validate the generalizability of our proposed approach.

CRediT authorship contribution statement

Guang Yang: Writing – original draft, Software, Data curation. **Yu Zhou:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Xiangyu Zhang:** Validation, Software, Data curation. **Xiang Chen:** Writing – review & editing, Validation. **Tingting Han:** Writing – review & editing, Validation. **Taolue Chen:** Writing – review & editing, Validation, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (NSFC, No. 62372232), the Postgraduate Research & Practice Innovation Program of Jiangsu Province, China (No. KYCX23_0396), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, China. T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University, China (KFKT2023A04).

Data availability

Data will be made available on request.

References

[1] J.D. Weisz, M. Muller, S. Houde, J. Richards, S.I. Ross, F. Martinez, M. Agarwal, K. Talamadupula, Perfection not required? human-ai partnerships in code translation, in: 26th International Conference on Intelligent User Interfaces, 2021, pp. 402–412.

[2] J.D. Weisz, M. Muller, S.I. Ross, F. Martinez, S. Houde, M. Agarwal, K. Talamadupula, J.T. Richards, Better together? an evaluation of ai-supported code translation, in: 27th International Conference on Intelligent User Interfaces, 2022, pp. 369–391.

[3] B. Roziere, M.-A. Lachaux, L. Chanussot, G. Lample, Unsupervised translation of programming languages, Adv. Neural Inf. Process. Syst. 33 (2020) 20601–20611.

[4] B. Roziere, J. Zhang, F. Charton, M. Harman, G. Synnaeve, G. Lample, Leveraging automated unit tests for unsupervised code translation, in: International Conference on Learning Representations.

[5] F. Liu, J. Li, L. Zhang, Syntax and domain aware model for unsupervised program translation, 2023, arXiv preprint arXiv:2302.03908.

[6] W.E. Zhang, Q.Z. Sheng, A. Alhazmi, C. Li, Adversarial attacks on deep-learning models in natural language processing: A survey, ACM Trans. Intell. Syst. Technol. (TIST) 11 (3) (2020) 1–41.

[7] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, Z. Jin, Generating adversarial examples for holding robustness of source code processing models, in: Proceedings of the AAAI Conference on Artificial Intelligence, 34, (01) 2020, pp. 1169–1176.

[8] M.R.I. Rabin, N.D. Bui, K. Wang, Y. Yu, L. Jiang, M.A. Alipour, On the generalizability of Neural Program Models with respect to semantic-preserving program transformations, Inf. Softw. Technol. 135 (2021) 106552.

[9] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, L. Zhang, 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, 2022, pp. 39–51.

[10] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, H. Gall, Adversarial robustness of deep code comment generation, ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (4) (2022) 1–30.

[11] Z. Yang, J. Shi, J. He, D. Lo, Natural attack for pre-trained models of code, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1482–1493.

[12] G. Yang, Y. Zhou, W. Yang, T. Yue, X. Chen, T. Chen, How important are good method names in neural code generation? A model robustness perspective, ACM Trans. Softw. Eng. Methodol. (2023) http://dx.doi.org/10.1145/3630010, Just Accepted.

[13] Github, GitHub copilot · your AI pair programmer, 2023, https://github.com/features/copilot. 2023-04-19.

[14] P. Vaithilingam, T. Zhang, E.L. Glassman, Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models, in: Chi Conference on Human Factors in Computing Systems Extended Abstracts, 2022, pp. 1–7.

[15] N. Grover, The ultimate review of GitHub copilot for language translation, 2023, https://medium.datadriveninvestor.com/the-ultimate-review-of-github-copilot-for-language-translation-5a32093eedfd. 2023-02-26.

[16] Y. Wang, W. Wang, S. Joty, S.C. Hoi, 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, 2021, pp. 8696–8708.

[17] S. Liu, B. Wu, X. Xie, G. Meng, Y. Liu, ContraBERT: Enhancing code pre-trained models via contrastive learning, 2023, arXiv preprint arXiv:2301.09072.

[18] 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, in: Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1).

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).

[20] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Trans. Softw. Eng. 30 (2) (2004) 126–139.

[21] W. Yuan, G. Neubig, P. Liu, Bartscore: Evaluating generated text as text generation, Adv. Neural Inf. Process. Syst. 34 (2021) 27263–27277.

[22] A. Hindle, E.T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: 2012 34th International Conference on Software Engineering, ICSE, IEEE, 2012, pp. 837–847.

[23] M.V. Pour, Z. Li, L. Ma, H. Hemmati, A search-based testing framework for deep neural networks of source code embedding, in: 2021 14th IEEE Conference on Software Testing, Verification and Validation, ICST, IEEE, 2021, pp. 36–46.

[24] Z. Dong, Q. Hu, Y. Guo, M. Cordy, M. Papadakis, Z. Zhang, Y. Le Traon, J. Zhao, MIXCODE: Enhancing Code Classification by Mixup-Based Data Augmentation.

[25] A. Jha, C.K. Reddy, Codeattack: Code-based adversarial attacks for pre-trained programming language models, in: Proceedings of the AAAI Conference on Artificial Intelligence, 37, (12) 2023, pp. 14892–14900.

[26] J. Henke, G. Ramakrishnan, Z. Wang, A. Albarghouth, S. Jha, T. Reps, Semantic robustness of models of source code, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2022, pp. 526–537.

[27] S. Chakraborty, T. Ahmed, Y. Ding, P.T. Devanbu, B. Ray, NatGen: generative pre-training by "naturalizing" source code, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 18–30.

[28] M. Wei, Y. Huang, J. Yang, J. Wang, S. Wang, Cocofuzzing: Testing neural code models with coverage-guided fuzzing, IEEE Trans. Reliab. (2022).

[29] P. Chen, Z. Li, Y. Wen, L. Liu, Generating adversarial source programs using important tokens-based structural transformations, in: 2022 26th International Conference on Engineering of Complex Computer Systems, ICECCS, IEEE, 2022, pp. 173–182.

[30] M.R.I. Rabin, M.A. Alipour, ProgramTransformer: A tool for generating semantically equivalent transformed programs, Softw. Impacts 14 (2022) 100429.

[31] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, et al., ReCode: Robustness evaluation of code generation models, 2022, arXiv preprint arXiv:2212.10264.

[32] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, Z. Jin, Towards robustness of deep program processing models—detection, estimation, and enhancement, ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (3) (2022) 1–40.

[33] J. Jia, S. Srikant, T. Mitrovska, C. Gan, S. Chang, S. Liu, U.-M. O'Reilly, CLAWSAT: Towards both robust and accurate code models, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2023, pp. 212–223.

[34] J. Tian, C. Wang, Z. Li, Y. Wen, Generating adversarial examples of source code classification models via Q-learning-based Markov decision process, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2021, pp. 807–818.

[35] 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, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

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

[37] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, A. Vladu, Towards deep learning models resistant to adversarial attacks, in: International Conference on Learning Representations.

[38] W.U. Ahmad, M.G.R. Tushar, S. Chakraborty, K.-W. Chang, Avatar: A parallel corpus for java-python program translation, 2021, arXiv preprint arXiv:2108.11590.

[39] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 2002, pp. 311–318.

[40] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, S. Ma, Codebleu: a method for automatic evaluation of code synthesis, 2020, arXiv preprint arXiv:2009.10297.
[41] Q. Liang, Z. Sun, Q. Zhu, W. Zhang, L. Yu, Y. Xiong, L. Zhang, Lyra: A benchmark for turducken-style code generation, 2021, arXiv preprint arXiv:2108.12144.
[42] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.d. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, 2021, arXiv preprint arXiv:2107.03374.
[43] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., GraphCodeBERT: Pre-training code representations with data flow, in: International Conference on Learning Representations.
[44] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, C. Xiong, CodeGen: An open large language model for code with multi-turn program synthesis, in: The Eleventh International Conference on Learning Representations, 2022.
[45] W. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, 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, 2021, pp. 2655–2668.
[46] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, J. Yin, 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), 2022, pp. 7212–7225.
[47] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X.E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, et al., Code llama: Open foundation models for code, 2023, arXiv preprint arXiv:2308.12950.
[48] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al., DeepSeek-coder: When the large language model meets programming–the rise of code intelligence, 2024, arXiv preprint arXiv:2401.14196.
[49] Q. Luo, H. Yu, X. Li, BAdam: A memory efficient full parameter training method for large language models, 2024, arXiv preprint arXiv:2404.02827.
[50] E. Shi, Y. Wang, H. Zhang, L. Du, S. Han, D. Zhang, H. Sun, Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond, 2023, arXiv preprint arXiv:2304.05216.
[51] A. Maćkiewicz, W. Ratajczak, Principal components analysis (PCA), Comput. Geosci. 19 (3) (1993) 303–342.
[52] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, et al., Emergent abilities of large language models, Trans. Mach. Learn. Res..
[53] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al., Training language models to follow instructions with human feedback, Adv. Neural Inf. Process. Syst. 35 (2022) 27730–27744.
[54] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al., The llama 3 herd of models, 2024, arXiv preprint arXiv:2407.21783.
[55] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al., Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025, arXiv preprint arXiv:2501.12948.
[56] P. Xue, L. Wu, C. Wang, X. Li, Z. Yang, R. Jin, Y. Zhang, J. Li, Y. Pei, Z. Shen, et al., Escalating LLM-based code translation benchmarking into the class-level era, 2024, arXiv preprint arXiv:2411.06145.
[57] A.T. Nguyen, T.T. Nguyen, T.N. Nguyen, Lexical statistical machine translation for language migration, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 651–654.
[58] S. Karaivanov, V. Raychev, M. Vechev, Phrase-based statistical translation of programming languages, in: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, 2014, pp. 173–184.
[59] K. An, N. Meng, E. Tilevich, Automatic inference of java-to-swift translation rules for porting mobile applications, in: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, 2018, pp. 180–190.
[60] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, Q. Wang, Mining API mapping for language migration, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, 2010, pp. 195–204.
[61] M.-A. Lachaux, B. Roziere, M. Szafraniec, G. Lample, Dobf: A deobfuscation pre-training objective for programming languages, Adv. Neural Inf. Process. Syst. 34 (2021) 14967–14979.

Guang Yang received the M.D. degree in computer technology from Nantong University, Nantong, in 2022. Then he is currently pursuing the Ph.D. degree at Nanjing University of Aeronautics and Astronautics, Nanjing. His research interest is AI4SE and he has authored or co-authored more than 30 papers in refereed journals or conferences, such as Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Journal of Systems and Software, International Conference on Software Maintenance and Evolution (ICSME), and International Conference on Software Analysis, Evolution and Reengineering (SANER). More information about him can be found at: https://ntdxyg.github.io/.

Yu Zhou is a full professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics (NUAA). He received his B.Sc. degree in 2004 and Ph.D. degree in 2009, both in Computer Science from Nanjing University China. Before joining NUAA in 2011, he conducted PostDoc research on software engineering at Politechnico di Milano, Italy. From 2015–2016, he visited the SEAL lab at University of Zurich Switzerland, where he is also an adjunct researcher. His current research interests mainly generative models for software engineering, software evolution analysis, mining software repositories, and reliability analysis. He has been supported by several national research programs in China. More information about him can be found at: https://csyuzhou.github.io/.

Xiangyu Zhang is currently pursuing a Master's degree at the College of Computer Science and Technology of Nanjing University of Aeronautics and Astronautics. His research interests include code generation and model interpretability.

Xiang Chen received the B.Sc. degree in the school of management from Xi'an Jiaotong University, China in 2002. Then he received his M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is currently an Associate Professor at the Department of Information Science and Technology, Nantong University, Nantong, China. He has authored or co-authored more than 120 papers in refereed journals or conferences, such as IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Information and Software Technology, Journal of Systems and Software, Journal of Software: Evolution and Process, Automated Software Engineering, Journal of Computer Science and Technology, International Conference on Software Engineering (ICSE), The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), International Conference Automated Software Engineering (ASE), International Conference on Software Maintenance and Evolution (ICSME), International Conference on Program Comprehension (ICPC), and International Conference on Software Analysis, Evolution and Reengineering (SANER). His research interests include software engineering, in particular software testing and maintenance, software repository mining, and empirical software engineering. He received two ACM SIGSOFT distinguished paper awards in ICSE 2021 and ICPC 2023. He is the editorial board member of Information and Software Technology. More information about him can be found at: https://smartse.github.io/index.html.

Taolue Chen received the Bachelor and Master degrees from Nanjing University, China, both in computer science. He was a junior researcher (OiO) at the Centrum Wiskunde & Informatica (CWI) and acquired the Ph.D. degree from the Vrije Universiteit Amsterdam, The Netherlands. He is currently a senior lecturer at the School of Computing and Mathematical Sciences, Birkbeck, University of London. He had been a postdoctoral researcher at University of Oxford (UK) and University of Twente (NL). His research area includes Software Engineering, Programming Language and Verification. His present research focus is on the border of software engineering and machine learning. He has published about 150 papers in journals and conferences such as POPL, LICS, CAV, ICSE, FSE, ASE, ISSTA, ETAPS (TACAS, FoSSaCS, ESOP, FASE), OOPSLA, NeurIPS, ICLR, EMNLP and IEEE TSE, ACM TOSEM, ACM TOCL. He won the Best Paper Award of SETTA'20, ACM SIGSOFT Distinguished Paper Award at ASE'24, the 1st Prize in the CCF Software Prototype Competition 2022 and the QF Strings (Single Query Track) at the International Satisfiability Modulo Theories Competition 2023. He has served editorial board or program committee for various international journals and conferences. More information about him can be found at https://chentaolue.github.io/.