Understanding the Robustness of Transformer-Based Code Intelligence via Code Transformation: Challenges and Opportunities

Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, *Fellow, IEEE*, Michael R. Lyu, *Life Fellow, IEEE*, and Zenglin Xu, *Senior Member, IEEE*

***Abstract*—Transformer-based models have demonstrated state-of-the-art performance in various intelligent coding tasks such as code comment generation and code completion. Previous studies show that deep learning models are sensitive to input variations, but few have systematically studied the robustness of Transformer under perturbed input code. In this work, we empirically study the effect of semantic-preserving code transformations on the performance of Transformers. Specifically, 27 and 24 code transformation strategies are implemented for two popular programming languages, Java and Python, respectively. To facilitating analysis, the strategies are grouped into five categories: block transformation, insertion / deletion transformation, grammatical statement transformation, grammatical token transformation, and identifier transformation. Experiments on three popular code intelligence tasks, including code completion, code summarization, and code search, demonstrate that insertion / deletion transformation and identifier transformation have the greatest impact on the performance of Transformers. Our results also suggest that Transformers based on abstract syntax trees (ASTs) show more robust performance than models based only on code sequences under most code transformations. Besides, the design of positional encoding can impact the robustness of Transformers under code transformations. We also investigate substantial code transformations at the strategy level to expand our study and explore other factors influencing the robustness of Transformers. Furthermore, we explore applications of code transformations. Based on our findings, we distill insights about the challenges and opportunities for Transformer-based code intelligence from various perspectives.**

***Index Terms*—Code intelligence, code transformation, transformer, robustness.**

I. Introduction

CODE intelligence aims to leverage AI techniques to enhance software developers' productivity during the development process [1], [2]. Over the past few years, deep neural networks (DNNs) have been continuously expanding their real-world applications to various code intelligence tasks [3], [4], [5], [6]. Among these advancements, Transformer-based models [7] have emerged as a focal point of academic research, largely due to the structural and syntactical similarities between source code and natural language [8]. Originally designed for capturing textual semantics through attention mechanisms [9], Transformer has proven highly effective in learning source code representations [10], [11], and has become a state-of-the-art architecture in several code intelligence tasks, including code completion [12], [13], code summarization [14], [15], and program repair [16].

Despite their success, DNNs have shown vulnerability to data changes [17], [18]. Previous studies highlighted that even adding minor perturbations to input data can readily trick DNNs [19], [20], revealing a lack of robustness to input variations [21], [22]. This issue has spurred research into understanding and improving the robustness of DNNs against such perturbations [23], [24], [25]. However, **developing such a robustness verification method for code intelligence models is challenging**. Directly applying the input perturbation techniques in natural language processing (NLP) or computer vision (CV) field [26], [27] is unreasonable, since the perturbation on source code must guarantee that the changed code follows syntax rules.

With the growing use of neural network models in code-related tasks, the robustness of code intelligence models has become a critical research focus to ensure their reliability and effectiveness [28], [29], [30], [31], [32], [33]. However, much of the research has focused on adversarial attacks, designed to induce errors or misclassifications in AI models through subtle input data manipulations. For instance, Yefet et al. [28] introduced a gradient-based optimization method to produce

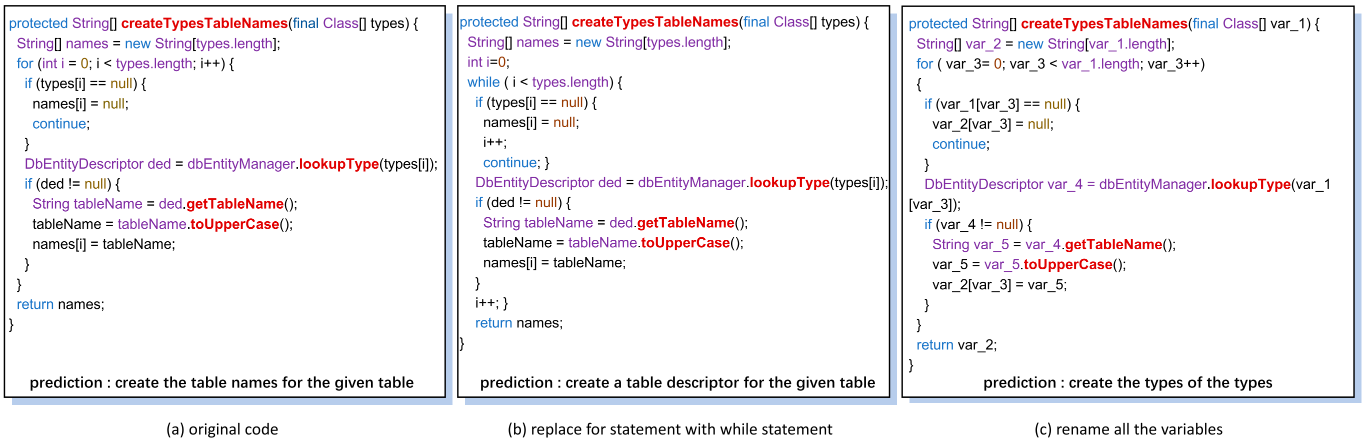
Received 4 July 2024; revised 5 December 2024; accepted 16 December 2024. Date of publication 16 January 2025; date of current version 13 February 2025. This work was supported in part by the National Natural Science Foundation of China under Project 62472126, in part by the Natural Science Foundation of Guangdong Province under Project 2023A1515011959, in part by Shenzhen-Hong Kong Jointly Funded Project (Category A, No. SGDX20230116091246007), in part by Shenzhen Basic Research under Project JCYJ20220531095214031, and in part by Shenzhen International Science and Technology Cooperation under Project GJHZ20220913143008015. Recommended for acceptance by T. Hall. *(Corresponding author: Cuiyun Gao.)*

Yaoxian Li, Shiyi Qi, Cuiyun Gao, and Zenglin Xu are with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen 518055, China (e-mail: gaocuiyun@hit.edu.cn; yaoxian0803@icloud.com; 21S051040@stu.hit.edu.cn; zenglin@gmail.com).

Yun Peng and Michael R. Lyu are with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong 999077, China (e-mail: ypeng@cse.cuhk.edu.hk; lyu@cse.cuhk.edu.hk).

David Lo is with School of Computing and Information Systems, Singapore Management University, Singapore 188065 (e-mail: davidlo@smu.edu.sg).

Digital Object Identifier 10.1109/TSE.2024.3524461



```
protected String[] createTypesTableNames(final Class[] types) {
  String[] names = new String[types.length];
  for (int i = 0; i < types.length; i++) {
    if (types[i] == null) {
      names[i] = null;
      continue;
    }
    DbEntityDescriptor ded = dbEntityManager.lookupType(types[i]);
    if (ded != null) {
      String tableName = ded.getTableName();
      tableName = tableName.toUpperCase();
      names[i] = tableName;
    }
  }
  return names;
}
```

prediction : create the table names for the given table

(a) original code

```
protected String[] createTypesTableNames(final Class[] types) {
  String[] names = new String[types.length];
  int i=0;
  while ( i < types.length) {
    if (types[i] == null) {
      names[i] = null;
      i++;
      continue; }
    DbEntityDescriptor ded = dbEntityManager.lookupType(types[i]);
    if (ded != null) {
      String tableName = ded.getTableName();
      tableName = tableName.toUpperCase();
      names[i] = tableName;
    }
    i++; }
  return names;
}
```

prediction : create a table descriptor for the given table

(b) replace for statement with while statement

```
protected String[] createTypesTableNames(final Class[] var_1) {
  String[] var_2 = new String[var_1.length];
  for ( var_3= 0; var_3 < var_1.length; var_3++)
  {
    if (var_1[var_3] == null) {
      var_2[var_3] = null;
      continue;
    }
    DbEntityDescriptor var_4 = dbEntityManager.lookupType(var_1
[var_3]);
    if (var_4 != null) {
      String var_5 = var_4.getTableName();
      var_5 = var_5.toUpperCase();
      var_2[var_3] = var_5;
    }
  }
  return var_2;
}
```

prediction : create the types of the types

(c) rename all the variables

Fig. 1. Examples of semantic-preserving code transformation. Figures from left to right represent the original code, semantically equivalent programs under for-to-while transformation, and variable rename transformation, respectively. The original reference is: create the table types for the given table.

adversarial attacks on code, effectively generating both targeted and non-targeted attacks. Zhang et al. [32] identified robust and non-robust features of DNNs and developed an identifier renaming algorithm (Metropolis-Hastings Modifier) to create adversarial examples in source code. Yang et al. [33] designed a black-box attack strategy that incorporates natural semantics for generating adversarial code examples. These studies have demonstrated the lack of robustness in code intelligence models. However, these works aim to cause the model to make different predictions via adversarial attacks, and focus exclusively on specific types of adversarial code. **Few studies have systematically examined the robustness of code intelligence models under perturbed input code.** Consequently, empirical investigations into these code intelligence models are essential, especially for the Transformer architecture, widely used for source code representation learning.

To bridge the above gaps, we propose several semantic-preserving code transformation strategies, and analyze the impact of code transformation on the performance of Transformer. Fig. 1 shows an example of semantically equivalent programs, where the code summaries are produced by the popular Transformer-based approach [14]. For the code listed in Fig. 1(a), we transform the *if statement* to equivalent *while statement*, as shown in Fig. 1(b), and conduct variable renaming, as shown in Fig. 1(c). However, the resulting summarizations of Transformer on the above three programs are radically different. Since the semantics of the original program are preserved, the model should produce the same prediction for the transformed programs as for the original program. This example suggests that (1) Transformer is not robust in code intelligence tasks when faced with semantic-preserving transformation, and (2) different code transformation strategies have different impacts on Transformer. Therefore, we aim to investigate whether Transformer can maintain performance under semantic-preserving code transformation, and the impact of different transformation strategies.

In this work, we empirically investigate the effect of semantic-preserving code transformation on the performance of Transformer. We first define modest changes in performance as a measure of the robustness of code intelligence mode. Then, we design and implement 27 and 24 semantic-preserving transformation strategies for Java and Python languages respectively, and categorize them into 5 types of strategies according to the scope of influence under the transformation: block transformation, insertion / deletion transformation, grammatical statement transformation, grammatical token transformation, and identifier transformation. Subsequently, we apply the transformed code to three popular code intelligence tasks: code completion (CC), code search (CS), and code summarization (CSM). Our selection accounts for the diversity of tasks—spanning code-to-code (CS), text-to-code (CS), and code-to-text (CSM) paradigms—to facilitate a comprehensive investigation of semantic-preserving transformations across various prediction objectives. For studying whether involving syntax information such as Abstract Syntax Trees (ASTs) is beneficial for improving the robustness of Transformer under code transformation, we classify the Transformer-based code intelligence models into two types according to the input: Seq-based Transformer and AST-based Transformer. The seq-based Transformer only considers sequences of code tokens as input; while the AST-based Transformer also involves parsed ASTs as input. Besides, the positional encoding is an essential component in Transformer [34], and has been proven effective in Transformer-based code models [8], [14], [35]. Consequently, this study also explores the effects of various positional encoding strategies on the robustness of Transformer models against code perturbations. Specifically, two widely-used positional encoding strategies, including absolute positional encoding [7] and relative positional encoding [36], are chosen for analysis. We aim at answering the following research questions in the work.

RQ1: How do different types of code transformations impact the performance of Seq-based Transformer

RQ2: Is AST helpful for reducing the impact of code transformations on the performance of Transformer? (AST-based Transformer)

The above research questions focus on the impacts on the robustness of Transformer models from the group perspective under code transformation and consider different positional encoding strategies. Additionally, we further explore the impact of code transformation from various perspectives, including the strategy scope, other factors affecting Transformer robustness under code transformation, and more applications of code transformation to study robustness.

RQ3: What is the impact of different code transformation strategies on Transformer? (Strategy scope)

RQ4: Are there other factors that affect the robustness of Transformer under code transformation?

RQ5: What impact does code transformation have on other models or tasks?

While answering each research question, we achieve some findings and summarize the key findings as follows.

- Code transformations such as insertion / deletion transformation and identifier transformation present the greatest impact on the performance of both seq-based and AST-based Transformers.
- Transformer based on ASTs shows more robust performance than the model based on only code sequence under most code transformations.
- The relative position encoding can improve the robustness of seq-based Transformer under most code transformations, but has no such benefit for the robustness of AST-based Transformer.

Based on the findings, we derive some insights about the challenges and opportunities that would benefit future research. For example, future work is expected to better exploit ASTs to boost the robustness of Transformer models under code transformation. Besides, we also encourage future work to explore more effective attention approaches or engage additional external knowledge to eliminate the distraction of insertion / deletion transformation. Furthermore, future work should eliminate the impact of identifiers during code representation learning, instead of relying on the semantics of identifiers.

The main contributions of this paper are summarized as follows:

- We empirically study the effect of semantic-preserving code transformations on the performance of Transformers in three popular code intelligence tasks.
- We design and implement 27 and 24 code transformation strategies for Java and Python, respectively.
- We study how different aspects impact the performance of Transformers, including input format, positional encoding strategy, sub-token, and model parameters.
- We further explore applications of code transformation, such as pre-trained models and code translation tasks.
- We achieve findings and implications that will benefit future research in the robustness of Transformer-based code intelligence tasks.

The rest of this paper is organized as follows. We present the background of Transformer and code intelligence tasks in Section II. The technical details of our code transformation strategies, evaluation, and study design are shown in Section III. Then, we present the experimental results and potential findings in Section IV. Based on the findings, we conclude some challenges and opportunities in Section V. We discuss threats to validity in Section VI. Finally, we review the literature related to our research in Section VII and conclude the work in Section VIII, respectively.

II. Background

A. Transformer and Positional Encoding

The Transformer architecture utilizes the conventional encoder-decoder structure [7] and comprises stacked Transformer blocks. Each block consists of a multi-head self-attention sub-layer, followed by a fully connected positional-wise feed-forward network sub-layer. The sub-layers are interconnected by residual connections [37] and layer normalization [38]. In contrast to the encoder, the decoder includes attention sub-layers that employ the key and value matrices from the encoder. In particular, positional encoding is a crucial component of the Transformer [34] and has demonstrated its effectiveness in code intelligence tasks [8]. In the following, we introduce two popular positional encoding strategies: absolute positional encoding [7] and relative positional encoding [36].

Absolute positional encoding. The original Transformer architecture employs positional encoding to account for the sequential nature of the input [7], specifically through the use of absolute positional encoding. With absolute positional encoding, each token in a sequence is assigned its own representation based on its location from 1 to the maximum sequence length and then mapped to a vector for the self-attention calculation. However, this encoding strategy cannot capture the ordering of the input tokens.

Relative positional encoding. To address the issue of capturing the ordering of input tokens, Shaw et al. [36] proposed the use of relative position encoding. Relative position encoding models the relationship between two elements through their distance in the input sequence and can encode the pairwise positional relationships between input elements. Unlike absolute position encoding, which only considers the position of a token relative to the beginning of the sequence, relative position encoding takes into account the distance between two tokens when calculating attention, resulting in a more effective and flexible approach.

B. Code intelligence Task

Code completion task. Code completion is a widely used feature in modern integrated development environments (IDEs) that aids programming [39]. Developers use code completion to predict expected code elements, such as class names and methods, based on the code provided at the point of prediction [13]. Token-level completion and statement-level completion are common code completion techniques [40]. In recent years, the success of Transformer boosts a series of code completion works based on Transformer-based models [41], [42]. For example, Alon et al. [43] proposed a structural language model based on Transformer, which leverages the syntax to model the code snippet as a tree to complete code. Liu et al. [44] pretrained a language model with a Transformer-based architecture and fine-tuned it for code completion. Kim et al. [12] proposed

TravTrans, a Transformer-based approach that leverages ASTs for code completion. In our experiment, we focus on token-level completion based on Transformer, where the task is to predict the next code token (n_i) based on the preceding code tokens [n_0,n_1,...,n_{i-1}].

Code summarization task. The objective of code summarization is to generate a natural language summary, such as a doc-string, for a given source code [6], [45]. Code summarization aids developers in comprehending the function and purpose of code without needing to read the code itself, thus saving them time [46]. In recent years, many works that applied deep learning models in code summarization achieved great success and became more and more popular [47], [48]. For instance, Iyer et al. [49] proposed an LSTM-based model with attention to generate code summaries for source code. Alon et al. [50] sampled and encoded random AST paths into LSTMs to generate summaries of source code. Ahmad et al. [14] utilized Transformer on code summarization for better mapping the source code to their corresponding natural language summaries. In our Transformer model for the code summarization task, given a dataset containing a set of programs C and their corresponding summaries S, the goal is to generate a summary $\tilde{s} = (s_0, s_1, \ldots, s_m)$ that maximizes the conditional likelihood $P(s|c)$ for a given code snippet $c = (c_0, c_1, \ldots, c_n)$ from C, where s is the corresponding summary in S.

Code search task. The goal of code search is to identify the most semantically related code snippet from a collection of code based on a given natural language query [51]. Traditional code search techniques have mainly relied on information retrieval methods [52], [53], whereas recent advancements have predominantly utilized deep neural networks [54], [55]. Gu et al. [5] introduced an RNN-based code search model that represents source code and natural language queries through joint embeddings. Additionally, Fan et al. [56] employed a self-attention network to construct a code representation network, which builds the semantic relationship between code snippets and queries. In our experiment, we focus on a Transformer-based neural code search model [57], which learns joint embeddings of the natural language query and the code snippet [56], which returns the expected ranking order of code snippets for a given natural language query [q_0, q_1,...,q_n].

III. Methodology

This section presents the approach of our study, which comprises three major phases: code transformation, model implementation, and robustness evaluation. In the code transformation phase, we introduce our semantic-preserving code transformation strategies, which are divided into five types based on the scope of the program's influence. In the model implementation phase, we implement Transformer models for three popular code intelligence tasks, respectively, and apply the transformed code to them. In the robustness evaluation phase, we first introduce our definition of robustness, then we analyze the performance of the models on transformed code and systematically evaluate the robustness of Transformer under code transformation.

A. Code Transformation

A semantic-preserving transformation maintains the input-output behavior of a program while altering its structure or representation [58]. This means that the transformed program computes the same function as the original program, albeit possibly in a different way. The semantic-preserving code transformation is implemented on AST, and consists of three phases: (1) we parse the source code into its AST using the standard compiler tool (e.g., tree-sitter[1] in our experiments); (2) we find transformable location and directly modify the structure of AST to the target code formation; (3) we convert the modified AST to a transformed source code. This process also needs to preserve the source code's original semantics and ensure the transformed code can be compilable and executable.

Table I presents all transformation strategies and the corresponding descriptions. To conduct a thorough investigation of the impact of transformed code on Transformer, we classify the code transformation strategies into five types according to the scope of influence under the transformation:

- Block transformation;
- Insertion / deletion transformation;
- Grammatical statement transformation;
- Grammatical token transformation;
- Identifier transformation.

Block transformation (denoted as T_B). This type refers to the code transformation that impacts the code at the block level, as shown in Fig. 2(b). The example illustrated in Fig. 1(b) is a block transformation, where the *for-statement* is transformed to the equivalent *while-statement*, and the loop conditional block has been restructured. This type contains seven code transformation strategies in total.

Insertion / deletion transformation (denoted as T_{ID}). This type of transformation is implemented at the statement level. It generally adds new statements or deletes existing ones without impacting other statements in the program. During implementation, only sub-trees are added or deleted at the AST level, as depicted in Fig. 2(c). For example, the *insert junk code transformation* will declare a new variable that is never used again, and such change will not affect other statements in the program (Shown in Fig. 4). This type includes seven code transformation strategies.

Grammatical statement transformation (denoted as T_{GS}). This type of transformation is also operated at the statement level. Different from T_{ID}, T_{GS} changes the statements and the associated AST nodes in the original code, as depicted in Fig. 2(d). For example, Fig. 4(d) presents an example of *expand unary operator* transformation. In this transformation, the statement *i++* is replaced by *i=i+1*, changing the statement. This type has ten code transformation strategies.

Grammatical token transformation (denoted as T_{GT}). This type of transformation changes the original code at the token level, and includes six code transformation strategies. As illustrated in Fig. 2(e), the transformation only affects the type and value of the associated AST node, leaving the structure unchanged. Note that this type of transformation does not involve

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

TABLE I
SEMANTIC-PRESERVING CODE TRANSFORMATION IN OUR EXPERIMENT

No.	Transformation strategy	Description of transformation	Java	Python
		Block transformation		
B-1	For to While	Replace the `for-statement` by equivalent `while-statement`	✓	✓
B-2	While to For	Replace the `while-statement` by equivalent `for-statement`	✓	✗
B-3	Elseif to Else If	Convert the `elseif` to `else if`. For example, `if(x==1){block1} else {if(x==2){block2}}` becomes `if(x==1){block1} else if (x==2){block2}`	✓	✓
B-4	Else If to Elseif	Convert the `else if` to `elseif`. For example, `if(x==1){block1} else if(x==2) {blcok2}` becomes `if (x==1) {block1} else { if(x==2) {block2} }`	✓	✓
B-5	If-Else Swap	Use the logical NOT operator to change the condition of the `if-statement` and exchange the block of `if` and `else`. For example, `if(x==0){block1} else {block2}` becomes `if !(x==0) {block2} else {block1}`	✓	✓
B-6	Decompose Complex If	Split the `if-statement` if the condition of `if-statement` has logical operator (e.g., `&&`)	✓	✓
B-7	Extract Function	Move the variable initialization statement to generate a new function, and then call the function, for example, `z=x+y` becomes `def func(x,y): return x+y z=func(x,y)`	✗	✓
		Insertion / deletion transformation		
ID-1	Insert Comments	Insert comments not related to the source code	✓	✓
ID-2	Insert Junk Code	Add code that not related to the source code	✓	✓
ID-3	Append Return Statement	Add a return statement at the end of the source code that returns the default value	✓	✓
ID-4	Import Unrelated Library	Import libraries that unrelated to source code	✓	✓
ID-5	Remove Comments	Remove all the comments from source code	✓	✓
ID-6	Replace Print with Pass	Replace the print statement without calculations by empty statement, for example, replace `print` by `pass`	✓	✓
ID-7	Delete Unused Variable	Remove the variable declaration statement if the variable is never used	✓	✓
		Grammatical statement transformation		
GS-1	Refactor Return Statement	If the `return-statement` returns an integer literal, we will declare a variable and return it	✓	✓
GS-2	Internalize For Loop Declaration	Move variable declaration into `for-statement`. For example, `int i; for(i=0;i<10;i++)` becomes `for(int i=0;i<10;i++)`	✓	✗
GS-3	Externalize For Loop Declaration	Move the variable declaration out of for statement	✓	✗
GS-4	Separate Declaration and Initialization	Split the variable declaration and initialization, for example, `int i=0` becomes `int i; i=0`	✓	✗
GS-5	Wrap with Logical NOT	Add the logical NOT operator and use the opposite comparison operator, for example, `x<y` becomes `!(y>=x)`	✓	✓
GS-6	Reverse Comparison Operator	Use the opposite comparison operator, for example, `x<y` becomes `y>x`	✓	✓
GS-7	Explicitize Assignment Operator	Change the argument assignment operator to the assignment operator, for example, `x+=1` becomes `x=x+1`	✓	✓
GS-8	Expand Unary Operator	Change the increment or decrease operator. For instance, `x++` becomes `x=x+1`	✓	✗
GS-9	Encapsulate in Curly Braces	Add a curly brace to a single statement and then generate a new compound statement	✓	✗
GS-10	Remove Redundant Braces	Delete curly of the compound if the compound statement has only a single statement	✓	✗
		Grammatical token transformation		
GT-1	Boolean to Integer	Replace TRUE or FALSE by 1 or 0	✗	✓
GT-2	Integer to Boolean	Replace 1 or 0 by TRUE or FALSE	✗	✓
GT-3	Promote Integral Type	Replace the integral type by a higher type, for example, replace the `int` by `long`	✓	✓
GT-4	Promote Floating Type	Replace the integral type by floating type, or replace the float type by double type	✓	✓
GT-5	Refactor Input API	Change the API for reading input	✗	✓
GT-6	Refactor Output API	Change the API for writing output	✓	✓
		Identifier transformation		
I-1	Rename Function and Class	Rename the function name and class name	✓	✓
I-2	Rename Variable	Rename the variable name	✓	✓

identifiers. For instance, Fig. 4(e) presents an example of the *promote integral type transformation*, which converts the data type of variable *sum* from *int* to *long*.

Identifier transformation (denoted as T_{I}). This type of transformation is also implemented at the token level but mainly operates on identifiers, as illustrated in Fig. 2(f). The type includes two transformation strategies, including *function rename transformation* and *variable rename transformation*. For example, the *variable rename transformation* renames the identifiers (variable name) with placeholders such as var1 and var2 as shown in Fig. 4.

Additionally, some strategies cannot be implemented for both languages considering the language-specific characteristics. For example, the Python language does not support the increment and decrement operators, so the *change the unary operator transformation* strategy is only allowed in Java. The Java language treats Boolean as a unique data type with two distinct values: True and False, so the *bool to int transformation* strategy is only applicable for Python.

B. Model Implementation

1) Datasets and Pre-Processing: In our experiment, we selected datasets for Java and Python, two widely used programming languages, from CodeSearchNet [51] for evaluation. CodeSearchNet is a collection of large datasets with code-document pairs from open-source projects on GitHub, which is widely used in code intelligence tasks [59], [60]. Detailed data statistics are illustrated in Table II. The subject data consist of 165K / 5K / 11K training / validation / test code snippets for Java and 252K / 14K / 15K for Python, respectively. For facilitating analysis, we parse the code snippets into ASTs, and

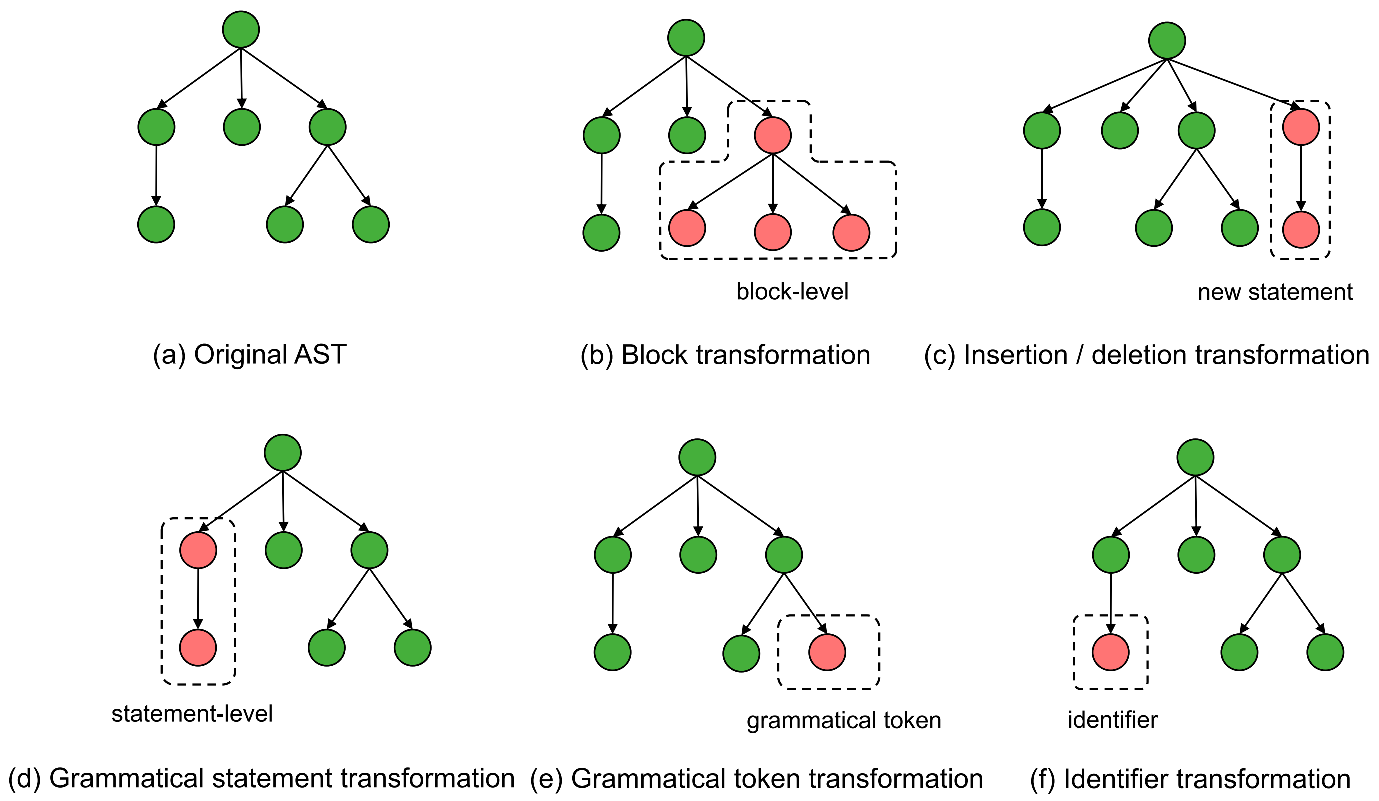

block-level
new statement
(a) Original AST
(b) Block transformation
(c) Insertion / deletion transformation
statement-level
grammatical token
identifier
(d) Grammatical statement transformation
(e) Grammatical token transformation
(f) Identifier transformation

Fig. 2. Example of the code transformation. (a) is an AST schematic, and (b)-(f) illustrate the different structure changes at the AST level with different code transformations.

TABLE II
STATISTICS OF EXPERIMENTAL DATA

Language	Train	Valid	Test
Python	251,820	13,914	14,918
Java	164,923	5,183	10,955

traverse the ASTs into sequences of tokens as input in depth-first-search order. We do not split sub-tokens to avoid the one-to-one relationship between the AST nodes and values being broken and resulting in inconsistent evaluation for the seq-based Transformer and AST-based Transformer [8]. Besides, considering that not all nodes in AST have associated values, we associate the <empty> value with nodes that do not have values, so that each node in ASTs has both and type value following the procedure done by Li et al. [61].

2) Transformer Implementation: In this work, we consider three code intelligence tasks: code completion, code summarization, and code search. We elaborate on the major implementation of the three tasks in the following.

Code completion. Our experiment focuses on token-level completion and utilizes the setup, metrics, and implementation of Transformer described in [12]. We use only a Transformer decoder to generate predictions. For the seq-based Transformer, the prediction object is the next code token (n_i) based on the previous code tokens [n_0,n_1,...,n_{i-1}]. The AST-based Transformer only focuses on value prediction rather than predicting all nodes for the AST, which is a one-to-one match with the seq-based model. Additionally, we split code sequences that are over 500 characters in length following [8] and set the vocabulary size to 100K for both seq-based and AST-based Transformers. This ensures consistency in our methodology and approach to training and evaluation.

Code summarization. For our experiments, we utilized the Transformer implementation from [14]. However, we did not split sub-tokens following Chirkova et al. [8] to maintain the one-to-one relationship between AST nodes and values. In this implementation, an encoder was used to obtain code representations, while a decoder generated the summary of the code. Additionally, we set the maximum lengths of code and code summaries to 400 and 300, respectively, as done in [14]. The vocabulary sizes were also set to 50K and 30K, respectively, following the same work. These settings provide consistency in our approach and facilitate the comparison of our results with previous works.

Code search. In our experiments, we implement a Transformer architecture for the code search task, which is based on the work of [62]. Unlike previous implementations, our model only trains a Transformer encoder to match the embeddings between natural language and code. We process the dataset following the strategy of Evangelos et al. [62]. For instance, we filter non-ASCII tokens and replace symbols with their English names (e.g., the symbol + in code token is replaced by `addoperator`, and the symbol = is replaced by `assign-operator`). Moreover, we filter out code documents that are less than 2 in length and set the maximum length of code at 256. The vocabulary sizes of code and natural language are set at 50K and 10K, respectively.

Hyper-parameters. The hyperparameters utilized in our experiments are based on Transformer implementations [12], [14], [62]. We describe the major hyperparameters for our code completion, code summarization, and code search tasks. Our Transformer models consist of 6 layers, with each layer having 6 heads and a dimension of 512. We set the maximum distance of relative attention to 32 for all tasks. We use Adam optimizer with an initial learning rate of 0.0001 and a batch size of 32 / 32 / 128 for code completion, summarization, and search, respectively. The number of epochs for training the models is 15 / 20 / 100 for code completion, summarization, and search, respectively. We train all models on 4 Nvidia Tesla V100 GPUs, each with 32GB of memory. For detailed implementations of all three tasks, including preprocessing scripts, please refer to our GitHub repository[2].

C. Robustness Evaluation

1) Robustness Definition: Several studies [30], [32], [33] have explored the robustness of code intelligence models from the perspective of adversarial attacks, focusing primarily on the task of function name prediction due to its binary evaluation metric of True or False. However, our investigation focuses on whether code intelligence models can maintain their performance and effectiveness despite variations in input data, rather than generating adversarial code that can lead to incorrect model predictions. Moreover, some code intelligence tasks are inappropriate for assessing the impact of code transformation using binary evaluation metrics. For instance, the exact match with ground-truth is unsuitable for evaluating the quality of generated code summaries in the code summarization task [63]. Therefore, we define **modest changes in performance** as a measure of the robustness of code intelligence models, which pertains to their ability to maintain performance when exposed to semantic-preserving transformed code. We employ this type of transformed code to investigate the model's robustness in handling code variations.

Let $P(M)$ be the performance of the model on the original subset of data M, and $P(M')$ be the performance on a transformed subset M', where the transformations are semantically preserved. For each code transformation strategy, we use ΔP to define the relative performance change for the code intelligence model that performs a specified task.

$$\Delta P = \frac{|P(M) - P(M')|}{P(M)}$$

From the group perspective, the performance change resulting from a code transformation group is calculated as the average change. This is determined by dividing the sum of all individual changes by the number of transformations in the group, which can be expressed as follows:

$$\Delta P_{group} = \frac{1}{n}\sum_{i=1}^{n}\Delta P_i$$

2) Evaluation Criteria of Code Transformation: One of the primary research objectives of our study is to investigate how various code transformation strategies impact the performance of code intelligence models. However, comparing the effects across different transformations presents three main challenges.

- **Selective applicability of code transformations.** The code suitable for different transformations varies significantly. Not all strategies can be applied to every piece of code; therefore, we only transform a subset of the code in the test set for each strategy. For instance, the *block transformation* cannot be applied if the original code lacks for/if/while statements, so we only transform code with block-related statements.
- **Language-specific transformation strategies.** Not all transformation strategies are applicable to all programming languages. For example, the *bool-to-int* strategy is specific to the Python language.
- **Complexity in assessing contributions.** The amount of transformable code for each transformation significantly influences model performance, complicating the assessment of a transformation's contribution. It is challenging to determine whether the impact is due to semantic changes or the volume of transformed code.

To address these challenges and accurately measure the contribution of code transformations, we have refined our evaluation approach. Our training and validation processes are aligned with the Transformer implementations for our code intelligence tasks. During the evaluation phase, we assess the impact of code transformation strategies on model performance by focusing solely on the transformable code, rather than the entire codebase in the testing dataset. In the open source repository[3], we provide details on the number of transformable code instances for each strategy and language. For example, there are 1,086 transformable code instances for the *for-statement* transformation, allowing us to directly compare the model performance on these instances before and after the transformation. This method ensures a more accurate evaluation by acknowledging the limitations of the applicability of the strategy.

Additionally, we need to establish uniform evaluation criteria for each transformation strategy. For the *block transformation*, *grammatical statement transformation*, *grammatical token transformation*, and *identifier transformation*, we transform all transformable parts of the given code. For example, if an original code contains multiple for-statements, we convert all of them into equivalent while-statements. For the textitinsertion / deletion transformation, we limit the number of transformation iterations to prevent potential endless usage. In the

[2]https://github.com/monsterLee599/CodeTrans-Transformer

[3]https://github.com/monsterLee599/CodeTrans-Transformer/tree/main/codetrans

ID-1/4 strategies, we set the transformation iteration limit to five, meaning we insert 5 string-format comments or unrelated libraries at the beginning of the given code. For the ID-2 strategy, we insert five pieces of junk code randomly into the source code, designed to ensure that they are never executed. The criteria for junk code stipulate that the inserted statements, such as a new variable declaration or a non-executing loop (e.g., for i in range(0): i), will never be executed.

To minimize performance bias, we run each experiment three times and report the average results. This approach helps to ensure that the reported results are reliable and robust.

3) Evaluation Metrics: We evaluate the source code summarization performance using three metrics: BLEU, METEOR, and ROUGE-L. For code search and code completion tasks, we use MRR as the evaluation metric.

BLEU is a widely-used metric in natural language processing and software engineering fields to evaluate the quality of generated texts, e.g., machine translation, code comment generation, and code commit message generation [64]. It computes the frequencies of the co-occurrence of n-grams between the ground truth $\hat{y}$ and the generated sequence y to judge the similarity:

$$\text{BLEU-N} = \mathrm{b}(\mathrm{y}, \hat{\mathrm{y}}) \cdot \exp\left(\sum_{n=1}^{N} \beta_n \log p_n(y, \hat{y})\right),$$

where $\mathrm{b}(\mathrm{y}, \hat{\mathrm{y}})$ indicates the brevity penalty, and $p_n(y, \hat{y})$ and β_n represent the geometric average of the modified n-gram precision and the weighting parameter, respectively.

ROUGE-L is commonly used in natural language translation [65], and is an F-measure based on the Longest Common Subsequence (LCS) between candidate and target sequences, where the LCS is a set of words appearing in the two sequences in the same order.

$$ROUGE\text{-}L = \frac{\left(1 + \beta^2\right) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}},$$

where $R_{lcs} = \frac{LCS(X,Y)}{len(Y)}$ and $P_{lcs} = \frac{LCS(X,Y)}{len(X)}$. X and Y denote the candidate sequence and reference sequence, respectively. $LCS(X, Y)$ represents the length of the longest common sub-sequence between X and Y.

Meteor is an evaluation metric proposed based on BLEU [66]. It introduces synonym, stem, and other information to replace the precise matching in BLEU, and strengthens the role of recall in automatic evaluation.

MRR is the average of the reciprocal rank of results of a set of queries [67]. The reciprocal rank of a query is the inverse of the rank of the first hit result.

$$MRR = \frac{1}{N} \sum_{n=1}^{N} \left(\frac{1}{rank_i}\right),$$

where N is the total number of targets (tokens for code completion and code snippet for code search) in the data pool and $rank_i$ represents the position of the i-th true target in the ranking results.

IV. Research Questions and Result Analysis

Our experimental study aims to answer the following research questions:

RQ1: How do different types of code transformations impact the performance of Transformer? (Seq-based Transformer)

RQ2: Is AST helpful for reducing the impact of code transformations on the performance of Transformer in the type scope? (AST-based Transformer)

RQ3: What is the impact of different code transformation strategies on Transformer? (Strategy scope)

RQ4: Are there other factors that affect the robustness of Transformer under code transformation?

- **RQ4.1:** Does the approach of AST traversal have an impact on Transformer robustness?
- **RQ4.2:** Does the pattern of token embeddings affect the robustness of the Transformer?
- **RQ4.3:** What is the impact of different parameters on the Transformer's robustness?

RQ5: What impact does code transformation have on other models or tasks?

- **RQ5.1:** How do different types of code transformations impact the performance of pre-training model?
- **RQ5.2:** What is the impact of code transformations on another task?

RQ1 aims to pinpoint the code transformations that exert the most substantial influence on the robustness of the Transformer. RQ2 investigates the utility of abstract syntax trees (AST) in enhancing the robustness of the Transformer under varying code transformations. RQ1 and RQ2 also consider the effects of different positional encoding strategies. Building on the insights gained from RQ1 and RQ2, RQ3 explores the impact of individual code transformation strategies. The study further investigates factors that mitigate the impact of code transformations on the Transformer's robustness, examining various AST traversal methods, token embedding patterns, and model parameters in RQ4.1, RQ4.2, and RQ4.3, respectively. Lastly, RQ5.1 and RQ5.2 evaluate the potential of the code transformation strategies by fine-tuning pre-training models for two code intelligence tasks and assessing the impact of code transformations on the code translation task, respectively.

A. Answer to RQ1: Impact on Seq-Based Transformer

In this section, we compare the performance of Transformer before and after different code transformations for three different tasks. Tables III to V present the overall results of code completion, code search, and code summarization, respectively.

1) Different Types of Code Transformation on Code Sequence: In this section, we analyze the effects of different types of code transformation on the performance of seq-based Transformer. We observe that seq-based Transformer's performance is affected to varying degrees by different types of code transformations. We elaborate on the detailed impact of different types of code transformation in the following.

TABLE III
Results of Code Transformation on the Performance of Seq-Based Transformer for the Code Completion Task. Column "Pos." Represents the Position Encoding Strategy, While "Abs." and "Rel." Represent the Absolute/Relative Position Encoding, Respectively. Column "w. t." and "w/o t." Represent the Results With and Without Code Transformation. Different Rows Represent the Results of Different Types of Code Transformation, While the Bottom Portion T_{ALL} Presents the Average Results. The Red Color Indicates the Degree of Decrease

			MRR (Java)			MRR (Python)	
Type	Pos.	w/o t.	w. t.	imp. (%)	w/o t.	w. t.	imp. (%)
T_{B}	abs.	0.7243	0.7185	↓ 0.81	0.6581	0.6539	↓ 0.65
	rel.	0.7343	0.7290	↓ 0.72	0.6708	0.6672	↓ 0.54
T_{ID}	abs.	0.7119	0.6107	↓ 14.22	0.6583	0.5503	↓ 16.40
	rel.	0.7208	0.6758	↓ 6.24	0.6698	0.6385	↓ 4.67
T_{GS}	abs.	0.7201	0.7082	↓ 1.66	0.6488	0.6377	↓ 1.72
	rel.	0.7304	0.7186	↓ 1.61	0.6614	0.6502	↓ 1.69
T_{GT}	abs.	0.7180	0.7107	↓ 1.02	0.6509	0.6301	↓ 3.19
	rel.	0.7277	0.7191	↓ 1.18	0.6637	0.6565	↓ 1.08
T_{I}	abs.	0.7169	0.6485	↓ 9.54	0.6594	0.6019	↓ 8.73
	rel.	0.7255	0.6544	↓ 9.81	0.6703	0.6136	↓ 8.46
T_{all}	abs.	0.7182	0.6793	↓ 5.42	0.6551	0.6148	↓ 6.16
	rel.	0.7036	0.6777	↓ 3.67	0.6672	0.6452	↓ 3.30

TABLE IV
Results of Code Transformation on the Performance of Seq-Based Transformer for the Code Search Task. The Red and Green Colors Indicate the Degree of Decrease and Increase, Respectively

			MRR (Java)			MRR (Python)	
Type	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
T_{B}	abs.	0.4356	0.4333	↓0.53	0.3795	0.3781	↓0.37
	rel.	0.4322	0.4337	↑0.36	0.3604	0.3655	↑1.40
T_{ID}	abs.	0.3976	0.2392	↓39.85	0.3945	0.2626	↓33.43
	rel.	0.4190	0.2612	↓37.66	0.3345	0.2132	↓36.27
T_{GS}	abs.	0.4209	0.4221	↑0.29	0.3868	0.3851	↓0.43
	rel.	0.4350	0.4367	↑0.39	0.3543	0.3606	↑1.76
T_{GT}	abs.	0.4404	0.4397	↓0.16	0.3764	0.3350	↓11.00
	rel.	0.4496	0.4484	↓0.27	0.3535	0.3173	↓10.22
T_{I}	abs.	0.4363	0.2522	↓42.21	0.4088	0.2557	↓37.44
	rel.	0.4427	0.2586	↓41.59	0.3974	0.2434	↓38.76
T_{all}	abs.	0.4261	0.3573	↓16.16	0.3892	0.3233	↓16.93
	rel.	0.4357	0.3677	↓15.60	0.3600	0.3000	↓16.68

Block transformation (T_{B}). As shown in Tables III–V, we observe that Transformer demonstrates robust performance under *block transformation* in the code intelligence tasks studied. For example, the MRR values for the code completion task just decrease by 0.81% and 0.65% for Java and Python, respectively (seen in Table III).

Insertion / deletion transformation (T_{ID}). From Tables III–V, we observe that *insertion / deletion transformation* has a substantial impact on Transformer on the studied code intelligence tasks. For example, the decrease of Transformer on the code search task is from 33.43% to 39.85% (seen in Table IV). When generating Java's code summary, the BLEU, ROUGE-L, and METEOR values decrease by 5.64%, 7.81%, and 14.18%, respectively (seen in Table V).

Grammatical statement transformation (T_{GS}). We find that seq-based Transformer shows robust performance under *grammatical statement transformation*. For instance, the impact of this type of code transformation on the code search task is 0.29% and -0.43% for Java and Python (seen in Table IV), respectively.

Grammatical token transformation (T_{GT}). From the experimental results, we observe that the *grammatical token transformation* has a slight influence on all tasks. For example, the MRR score has a decrease of 1.02% and 3.19% respectively for Java and Python on the code completion task, respectively (seen in Table III).

Identifier transformation (T_{I}). We observe that this type of code transformation has a substantial impact on the studied code intelligence tasks. For instance, the MRR score has a decrease of 42.21% and 38.22% for Java and Python under *identifier transformation* in the code search task, respectively (seen in Table IV).

Finding 1: Seq-based Transformer's performance is affected to varying degrees by different types of code transformations.

Based on the above analysis, we achieve that seq-based Transformer shows robust performance under *block transformation*, *grammatical statement transformation* and *grammatical token transformation*, but suffers from obvious performance degradation under *insertion / deletion transformation* and *identifier transformation*. For example, in code completion task, the decreases of MRR score for Java under *insertion / deletion transformation* and *identifier transformation* are 14.22% and 9.54%, respectively, while the decreases caused by *block transformation*, *grammatical statement transformation* and *grammatical token transformation* are 0.81%, 1.66%, and 1.02%, respectively.

Finding 2: The insertion / deletion transformation and identifier transformation present greatest impact on the performance of seq-based Transformer.

2) Absolute Position v.s. Relative Position: The section looks into whether position encoding impacts Transformer's performance under code transformation. From Tables III–V, we find that the Transformer with relative position encoding (Transformer$_{\mathrm{Rel}}$) shows a better robust performance compared to that with absolute position encoding (Transformer$_{\mathrm{Abs}}$) on three code intelligence tasks. For example, the overall MRR score of Transformer$_{\mathrm{Abs}}$ has a decrease of 5.42% and 6.16% for Java and Python in the code completion task, while the decrease of Transformer$_{\mathrm{Rel}}$ is 3.67% and 3.30%, respectively. More specifically, we find that the influence of relative position encoding is more obvious under *insertion / deletion transformation* than other types of code transformation on the code completion task. For example, the Transformer$_{\mathrm{Abs}}$ and Transformer$_{\mathrm{Rel}}$' MRR values under *insertion / deletion transformation* decrease by 14.22% and 6.24% for Java and 16.40% and 4.67% for Python, respectively, while the decreases under *grammatical statement transformation* are are 1.66% and 1.61% for Java and 1.72% v.s. 1.69% for Python, respectively

TABLE V
RESULTS OF CODE TRANSFORMATION ON THE PERFORMANCE OF SEQ-BASED TRANSFORMER FOR THE CODE SUMMARIZATION TASK. THE ABOVE PART PRESENTS THE RESULTS OF TRANSFORMER WITH ABSOLUTE/RELATIVE POSITION ENCODING STRATEGIES ("ABS." AND "REL.") FOR JAVA, AND THE BELOW PART PRESENTS RESULTS FOR PYTHON. THE RED AND GREEN COLORS INDICATE THE DEGREE OF DECREASE AND INCREASE, RESPECTIVELY

Java		BLEU			ROUGE-L			METEOR		
Type	Pos.	w/o. t.	w. t	Imp.(%)	w/o. t.	w. t	Imp.(%)	w/o. t.	w. t	Imp.(%)
T_{B}	abs.	11.73	11.83	↑ 0.81	21.82	21.93	↑ 0.50	6.08	6.30	↑ 3.74
	rel.	12.09	12.08	↓ 0.11	22.36	22.31	↓ 0.22	6.93	6.92	↓ 0.03
T_{ID}	abs.	13.36	12.61	↓ 5.64	22.82	21.04	↓ 7.81	6.84	5.87	↓ 14.18
	rel.	13.33	12.69	↓ 4.76	23.33	21.59	↓ 7.46	7.49	6.29	↓ 15.98
T_{GS}	abs.	12.73	12.68	↓ 0.40	23.30	23.19	↓ 0.45	6.93	7.02	↑ 1.35
	rel.	12.93	12.93	↓ 0.03	23.55	23.50	↓ 0.21	7.53	7.58	↑ 0.72
T_{GT}	abs.	12.92	12.83	↓ 0.63	22.94	22.82	↓ 0.50	6.78	6.86	↑ 1.17
	rel.	13.18	13.16	↓ 0.16	23.75	23.68	↓ 0.31	7.63	7.57	↓ 0.78
T_{I}	abs.	13.80	12.69	↓ 8.02	24.44	21.77	↓ 10.95	7.53	5.04	↓ 33.10
	rel.	13.70	12.86	↓ 6.13	24.93	22.48	↓ 9.84	8.29	6.42	↓ 22.61
T_{all}	abs.	12.91	12.53	↓ 2.94	23.06	22.15	↓ 3.96	6.83	6.22	↓ 8.96
	rel.	13.05	12.75	↓ 2.32	23.59	22.71	↓ 3.70	7.57	6.96	↓ 8.13
Python		BLEU			ROUGE-L			METEOR		
T_{B}	abs.	12.83	12.72	↓ 0.90	21.12	21.00	↓ 0.53	5.60	5.49	↓ 1.87
	rel.	13.49	13.37	↓ 0.93	22.46	22.26	↓ 0.88	6.76	6.58	↓ 2.73
T_{ID}	abs.	13.74	12.93	↓ 5.86	22.19	20.13	↓ 9.28	6.15	4.98	↓ 19.11
	rel.	14.18	13.60	↓ 4.07	23.05	21.57	↓ 6.42	7.16	6.20	↓ 13.37
T_{GS}	abs.	12.83	12.78	↓ 0.41	21.60	21.53	↓ 0.29	5.59	5.51	↓ 1.34
	rel.	13.45	13.40	↓ 0.40	22.90	22.79	↓ 0.49	7.08	6.95	↓ 1.77
T_{GT}	abs.	13.18	12.99	↓ 1.41	21.09	20.60	↓ 2.34	4.78	4.52	↓ 5.41
	rel.	13.71	13.58	↓ 0.94	22.40	22.07	↓ 1.50	6.42	6.29	↓ 1.91
T_{I}	abs.	13.84	13.17	↓ 4.83	23.06	21.41	↓ 7.13	6.52	5.37	↓ 17.71
	rel.	14.33	13.82	↓ 3.57	24.11	22.71	↓ 5.80	7.78	6.85	↓ 11.99
T_{all}	abs.	13.28	12.92	↓ 2.75	21.81	20.94	↓ 4.01	5.73	5.17	↓ 9.67
	rel.	13.83	13.55	↓ 2.02	22.98	22.28	↓ 3.07	7.04	6.57	↓ 6.60

(T_{ID} and T_{GS} in Table III) The robustness improvement may be attributed to that the relative position encoding enhances the relationship between the tokens with the adjacent preceding tokens [36], making Transformer's predictions more accurate on code completion.

Finding 3: Seq-based Transformer with relative position encoding shows more robust performance compared to that with absolute position encoding under most code transformations.

To sum up, seq-based Transformer shows robust performance under *block transformation*, *grammatical statement transformation* and *grammatical token transformation*. However, *insertion / deletion transformation* and *identifier transformation* have a substantial impact on Transformer's performance. Besides, the relative position encoding can improve the robustness of seq-based Transformer under most code transformations.

TABLE VI
RESULTS OF CODE TRANSFORMATION ON THE PERFORMANCE OF AST-BASED TRANSFORMER FOR THE CODE COMPLETION TASK. THE RED AND GREEN COLORS INDICATE THE DEGREE OF DECREASE AND INCREASE, RESPECTIVELY

		MRR (Java)			MRR (Python)		
Type	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
T_{B}	abs.	0.8030	0.8043	↑ 0.16	0.7630	0.7624	↓ 0.08
	rel.	0.8050	0.8066	↑ 0.20	0.7646	0.7644	↓ 0.01
T_{ID}	abs.	0.7879	0.7520	↓ 4.56	0.7570	0.7334	↓ 3.12
	rel.	0.7897	0.7542	↓ 4.50	0.7590	0.7388	↓ 2.66
T_{GS}	abs.	0.7996	0.7951	↓ 0.55	0.7554	0.7504	↓ 0.67
	rel.	0.8026	0.7983	↓ 0.53	0.7577	0.7526	↓ 0.67
T_{GT}	abs.	0.8001	0.7941	↓ 0.75	0.7531	0.7444	↓ 1.17
	rel.	0.8017	0.7961	↓ 0.70	0.7547	0.7477	↓ 0.92
T_{I}	abs.	0.7896	0.7201	↓ 8.81	0.7569	0.7098	↓ 6.23
	rel.	0.7916	0.7210	↓ 8.91	0.7589	0.7107	↓ 6.35
T_{all}	abs.	0.7960	0.7731	↓ 2.88	0.7571	0.7401	↓ 2.25
	rel.	0.7981	0.7752	↓ 2.60	0.7590	0.7429	↓ 2.12

B. Answer to RQ2: Impact on AST-Based Transformer

In this section, we compare the performance of AST-based Transformer before and after different code transformations for three different tasks. Tables VI–VIII present the overall results of code completion, code search, and code summarization on ASTs, respectively.

1) Different Types of Code Transformation on ASTs: In this section, we analyze the effects of different types of code transformation on the performance of AST-based Transformer. We observe that *insertion / deletion transformation* and *identifier transformation* also have a substantial impact on

TABLE VII
RESULTS OF CODE TRANSFORMATION ON THE PERFORMANCE OF AST-BASED TRANSFORMER FOR THE CODE SEARCH TASK. THE RED AND GREEN COLORS INDICATE THE DEGREE OF DECREASE AND INCREASE, RESPECTIVELY

		MRR (Java)			MRR (Python)		
Type	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
T_B	abs.	0.4242	0.4256	↑0.31	0.3981	0.3995	↑0.33
	rel.	0.4468	0.4471	↑0.06	0.4051	0.4041	↓0.25
T_{ID}	abs.	0.3978	0.3457	↓13.10	0.4110	0.2791	↓32.09
	rel.	0.4373	0.3797	↓13.17	0.3853	0.2650	↓31.24
T_{GS}	abs.	0.4076	0.4067	↓0.22	0.4137	0.4119	↓0.42
	rel.	0.4248	0.4241	↓0.17	0.4123	0.4115	↓0.19
T_{GT}	abs.	0.4393	0.4356	↓0.84	0.3967	0.3458	↓12.85
	rel.	0.4637	0.4610	↓0.59	0.3971	0.3435	↓13.50
T_I	abs.	0.4238	0.2421	↓42.86	0.4251	0.2647	↓37.74
	rel.	0.4501	0.2594	↓42.37	0.4298	0.2689	↓37.43
T_{all}	abs.	0.4185	0.3711	↓11.33	0.4089	0.3402	↓16.82
	rel.	0.4446	0.3943	↓11.31	0.4059	0.3386	↓16.59

AST-based Transformer. For example, the decrease of AST-based Transformer on predicting Java's code are 4.56% and 8.81% under *insertion / deletion transformation* and *identifier transformation*, respectively, while the decrease of that under *grammatical statement transformation* and *grammatical token transformation* are 0.55% and 0.75%, respectively. We elaborate on the detailed impact of ASTs on different types of code transformation in the following.

Block transformation (T_B) has a minor impact on AST-based Transformer's performance as Tables VI–VIII shows. And AST-based Transformer shows more robust performance than seq-based Transformer. For example, when it comes to generating Python's code summary, the values of BLEU, ROUGE-L, and METEOR decrease by 0.06%, 0.09%, and 0.44% on AST-based Transformer, while the decreases on seq-based Transformer are 0.90%, 0.53%, 1.87%, respectively.

Insertion / deletion transformation (T_{ID}). We find that incorporating ASTs into Transformer can increase the model's robustness under *insertion / deletion transformation* on the code completion task, Python's code summarization task, and Java's code search task. For example, the MRR score has a decrease of 14.22% and 16.40% for Java and Python on the seq-based Transformer (see Table III), while the decrease is 4.56% and 3.12% on the AST-based Transformer (see Table VI), respectively.

Grammatical statement transformation (T_{GS}) has a minor effect on AST-based Transformer. For instance, the MRR values decrease by 0.55% and 0.67%, respectively, when in Java's and Python's code completion.

Grammatical token transformation (T_{GT}). From the results, we also find that *grammatical token transformation* has a smaller influence on AST-based Transformer than seq-based Transformer. For example, the MRR score of AST-based Transformer under *grammatical token transformation* has a decrease of 0.75% and 1.17% for Java and Python in code completion task, respectively, while the decrease of seq-based Transformer is 1.02% and 3.19%, respectively.

Identifier transformation (T_I) also results in a large quality drop of the Transformer on the AST traverse. We take the code search task under *identifier transformation* as an example, AST-based Transformer has decreased MRR by 42.86% and 37.74% for Java and Python, respectively, while seq-based Transformer has decreased MRR by 42.21% and 37.44%. This result shows that the AST-based Transformer is also vulnerable to *identifier transformation* as the seq-based Transformer.

Compared to the seq-based Transformer, utilizing ASTs helps the Transformer maintain performance better. For example, the overall MRR score of the seq-based Transformer decreases by 5.42% on Java code completion (Table III), while the AST-based Transformer decreases by 2.88%. Similar results are seen in Python (6.16% vs. 2.25%). In the code search task, the overall performance decreases for the seq-based Transformer are 16.16% and 16.93% (Table IV) for Java and Python, while the decreases for the AST-based Transformer are 11.33% and 16.82% (Table VII), respectively.

Finding 4: Compared to seq-based Transformer, AST-based Transformer demonstrates more robust performance under most code transformations.

2) Absolute Position v.s. Relative Position on ASTs: This section looks into whether position encoding has an impact on the AST-based Transformer's performance under code transformation. From Tables VI–VIII, we find that relative position encoding does not make obvious improvement on AST-based Transformer's robustness under code transformations. For example, the decrease in overall MRR score on Java's code completion task is 2.88% and 2.60% for absolute and relative position encoding, while the decrease on Python is 2.25% and 2.12%, respectively. Similar results can be seen on the code search task (11.33% v.s. 11.31% for Java and 16.82% v.s. 16.59% for Python).

Finding 5: Relative position encoding does not evidently improve the robustness of AST-based Transformer under code transformation.

To sum up, the greatest impact of two types of code transformations on AST-based Transformer are also insertion / deletion transformation and identifier transformation. And AST-based Transformer performs more robustly compared to seq-based Transformer. Besides, relative position encoding does not evidently improve the robustness of AST-based Transformer when faced with code transformation.

C. Answer to RQ3: Impact in the Strategy Scope

In this section, we further investigate the effect of different code transformation strategies on the performance of three tasks from the strategy perspective. Since the majority of code transformation strategies have little effect on Transformer's performance, we only look into select notable code transformation results and provide a partial list of them. The complete results can be referred to the GitHub repository[4].

[4]https://github.com/monsterLee599/CodeTrans-Transformer

TABLE VIII
RESULTS OF CODE TRANSFORMATION ON THE PERFORMANCE OF AST-BASED TRANSFORMER FOR THE CODE SUMMARIZATION TASK. THE ABOVE PART PRESENTS THE RESULTS OF TRANSFORMER WITH ABSOLUTE/RELATIVE POSITION ENCODING STRATEGIES ("ABS." AND "REL.") FOR JAVA, AND THE BELOW PART PRESENTS RESULTS FOR PYTHON. THE RED AND GREEN COLORS INDICATE THE DEGREE OF DECREASE AND INCREASE, RESPECTIVELY

Java		BLEU			ROUGE-L			METEOR		
Type	Pos.	w/o. t.	w. t	Imp. (%)	w/o. t.	w. t	Imp. (%)	w/o. t.	w. t	Imp. (%)
T_{B}	abs.	11.89	11.89	$\rightarrow$ 0.00	21.85	21.81	$\downarrow$ 0.19	6.91	7.01	$\uparrow$ 1.39
	rel.	12.40	12.43	$\uparrow$ 0.22	22.92	22.92	$\downarrow$ 0.01	7.53	7.54	$\uparrow$ 0.09
T_{ID}	abs.	13.35	12.31	$\downarrow$ 7.78	22.69	19.91	$\downarrow$ 12.25	7.24	5.91	$\downarrow$ 18.30
	rel.	13.41	12.66	$\downarrow$ 5.61	23.26	20.89	$\downarrow$ 10.20	7.62	6.52	$\downarrow$ 14.39
T_{GS}	abs.	12.70	12.84	$\uparrow$ 1.14	23.06	23.32	$\uparrow$ 1.12	7.67	7.76	$\uparrow$ 1.18
	rel.	12.99	13.00	$\uparrow$ 0.12	23.40	23.31	$\downarrow$ 0.39	8.14	8.18	$\uparrow$ 0.43
T_{GT}	abs.	12.93	12.88	$\downarrow$ 0.39	23.03	22.76	$\downarrow$ 1.17	7.38	7.34	$\downarrow$ 0.52
	rel.	13.25	13.26	$\uparrow$ 0.08	23.65	23.52	$\downarrow$ 0.55	8.06	7.93	$\downarrow$ 1.56
T_{I}	abs.	13.94	12.92	$\downarrow$ 7.31	24.65	22.05	$\downarrow$ 10.53	8.24	6.77	$\downarrow$ 17.93
	rel.	13.82	13.06	$\downarrow$ 5.53	24.99	22.90	$\downarrow$ 8.36	8.66	7.47	$\downarrow$ 13.83
T_{all}	abs.	12.96	12.57	$\downarrow$ 3.03	23.05	21.97	$\downarrow$ 4.71	7.49	6.96	$\downarrow$ 7.09
	rel.	13.17	12.88	$\downarrow$ 2.22	23.64	22.71	$\downarrow$ 3.96	8.00	7.53	$\downarrow$ 5.94
Python		BLEU			ROUGE-L			METEOR		
Type	Pos.	w.o. t.	w. t	Imp.	w.o. t.	w. t	Imp.	w.o. t.	w. t	Imp.
T_{B}	abs.	12.51	12.50	$\downarrow$ 0.06	18.89	18.88	$\downarrow$ 0.09	4.85	4.83	$\downarrow$ 0.44
	rel.	13.11	13.05	$\downarrow$ 0.47	19.66	19.60	$\downarrow$ 0.29	5.94	5.94	$\uparrow$ 0.10
T_{ID}	abs.	12.87	12.64	$\downarrow$ 1.77	19.18	18.60	$\downarrow$ 3.00	5.05	4.62	$\downarrow$ 8.50
	rel.	13.77	13.10	$\downarrow$ 4.88	20.21	18.99	$\downarrow$ 6.04	6.17	5.49	$\downarrow$ 11.00
T_{GS}	abs.	12.39	12.37	$\downarrow$ 0.16	19.09	19.03	$\downarrow$ 0.34	4.97	4.90	$\downarrow$ 1.46
	rel.	12.97	12.93	$\downarrow$ 0.35	19.79	19.71	$\downarrow$ 0.41	5.97	5.97	$\downarrow$ 0.01
T_{GT}	abs.	12.76	12.72	$\downarrow$ 0.38	19.13	19.01	$\downarrow$ 0.68	4.74	4.78	$\uparrow$ 0.99
	rel.	13.41	13.31	$\downarrow$ 0.73	19.79	19.74	$\downarrow$ 0.26	5.68	5.59	$\downarrow$ 1.65
T_{I}	abs.	12.93	12.78	$\downarrow$ 1.16	19.69	19.25	$\downarrow$ 2.23	5.35	4.93	$\downarrow$ 7.79
	rel.	13.80	13.45	$\downarrow$ 2.56	21.06	20.13	$\downarrow$ 4.39	6.37	5.61	$\downarrow$ 11.98
T_{all}	abs.	12.69	12.60	$\downarrow$ 0.72	19.20	18.95	$\downarrow$ 1.28	4.99	4.81	$\downarrow$ 3.58
	rel.	13.41	13.17	$\downarrow$ 1.83	20.10	19.63	$\downarrow$ 2.32	6.03	5.72	$\downarrow$ 5.08

1) Effect of Different Insert Locations: During our experiments, we observe that the placement of junk code at different locations (*ID-2 front / middle / end*) impacts the performance of the sequence-based Transformer variably. The default implementation, *ID-2 middle*, involves randomly inserting junk code within the code. *ID-2 front* and *ID-2 end* involve inserting junk code at the beginning and end of the code, respectively. The results, as depicted in Table IX, show that the Transformer's performance is most adversely affected when junk code is inserted at the beginning of the code, compared to random or end placements. For instance, in the code search task, the performance decreases for Java are 42.67%, 25.31%, and 15.95% for *ID-2 front / middle / end*, respectively. In Python, the decreases are even more pronounced, at 88.19%, 14.78%, and 10.56% for the respective placements.

Furthermore, we observe that inserting junk code snippets (*ID-2*) at the beginning of the code similarly degrades the performance of the AST-based Transformer, just as with the seq-based Transformer. However, the AST-based Transformer exhibits greater robustness across different insertion points compared to the seq-based model. As illustrated in Table X, for the Java code search task, the performance decreases for the

TABLE IX
IMPACT OF DIFFERENT INSERT LOCATIONS OF JUNK CODE ON SEQ-BASED TRANSFORMER. CS AND CSM REPRESENT CODE SEARCH AND CODE SUMMARIZATION TASKS, RESPECTIVELY. THE "LOC." REPRESENTS THE INSERT LOCATIONS, AND MIDDLE / FRONT / END REPRESENT INSERTING JUNK CODE TO THE GIVEN CODE AT THE MIDDLE / AT THE FRONT / AT THE END, RESPECTIVELY. THE RED COLOR INDICATES THE DEGREE OF DECREASE

CS		MRR (Java)			MRR (Python)		
Loc.	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
front	abs.	0.4404	0.2525	$\downarrow$ 42.67	0.3901	0.0461	$\downarrow$ 88.19
	rel.	0.4461	0.2606	$\downarrow$ 41.57	0.4117	0.0564	$\downarrow$ 86.30
middle	abs.	0.4404	0.3290	$\downarrow$ 25.31	0.3901	0.3325	$\downarrow$ 14.78
	rel.	0.4461	0.3305	$\downarrow$ 25.91	0.4117	0.3324	$\downarrow$ 19.26
end	abs.	0.4404	0.3702	$\downarrow$ 15.95	0.3901	0.3489	$\downarrow$ 10.56
	rel.	0.4461	0.3748	$\downarrow$ 15.98	0.4117	0.3456	$\downarrow$ 16.05
CSM		BLEU (Java)			BLEU (Python)		
Loc.	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
front	abs.	13.80	12.07	$\downarrow$ 12.49	13.85	11.67	$\downarrow$ 15.76
	rel.	13.70	12.29	$\downarrow$ 10.31	14.35	12.43	$\downarrow$ 13.38
middle	abs.	13.80	12.13	$\downarrow$ 12.06	13.85	12.27	$\downarrow$ 11.41
	rel.	13.70	12.30	$\downarrow$ 10.27	14.35	13.11	$\downarrow$ 8.64
end	abs.	13.80	12.24	$\downarrow$ 11.28	13.85	12.47	$\downarrow$ 9.99
	rel.	13.70	12.42	$\downarrow$ 9.39	14.35	13.25	$\downarrow$ 7.64

TABLE X
IMPACT OF DIFFERENT INSERT LOCATIONS OF JUNK CODE ON AST-BASED TRANSFORMER. CS AND CSM REPRESENT CODE SEARCH AND CODE SUMMARIZATION TASKS, RESPECTIVELY. THE "LOC." REPRESENTS THE INSERT LOCATIONS, AND MIDDLE / FRONT / END REPRESENT INSERTING JUNK CODE TO THE GIVEN CODE AT THE MIDDLE / AT THE FRONT / AT THE END, RESPECTIVELY. THE RED COLOR INDICATES THE DEGREE OF DECREASE

CS		MRR (Java)			MRR (Python)		
Loc.	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
front	abs.	0.4184	0.3565	↓ 14.79	0.4364	0.0601	↓ 86.23
	rel.	0.4242	0.3448	↓ 18.72	0.4200	0.0389	↓ 90.74
middle	abs.	0.4184	0.3821	↓ 8.66	0.4364	0.3750	↓ 14.06
	rel.	0.4242	0.3826	↓ 9.81	0.4200	0.3564	↓ 15.13
end	abs.	0.4184	0.4009	↓ 4.17	0.4364	0.3981	↓ 8.78
	rel.	0.4242	0.4026	↓ 5.10	0.4200	0.3860	↓ 8.09
CSM		**BLEU (Java)**			**BLEU (Python)**		
Loc.	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
front	abs.	13.94	12.66	↓ 9.18	12.93	12.47	↓ 3.56
	rel.	13.82	12.47	↓ 9.77	13.81	12.42	↓ 10.09
middle	abs.	13.94	12.75	↓ 8.54	12.93	12.63	↓ 2.35
	rel.	13.82	12.68	↓ 8.25	13.81	13.08	↓ 5.28
end	abs.	13.94	12.86	↓ 7.75	12.93	12.66	↓ 2.11
	rel.	13.82	12.80	↓ 7.40	13.81	13.12	↓ 5.02

seq-based Transformer at *ID-2 front / middle / end* positions are 42.67%, 25.31%, and 15.95%, respectively (referenced in Table IX). In contrast, the reductions in performance for the AST-based Transformer at the same positions are considerably lesser, marked at 14.79%, 8.66%, and 4.17%, respectively.

Finding 6: Inserting junk code at the front of the given code can affect Transformer's performance more than inserting junk code at other locations.

The analysis of the results demonstrates that Transformers are particularly sensitive to information presented at the beginning of their input. This observation aligns with the concept of Primacy Bias, as discussed in [68], which refers to the tendency of models to better retain and utilize information appearing earlier in the input while struggling to effectively process information in the middle of long contexts. Placing junk code at the beginning disrupts this natural focus on early information, interfering with the model's typical processing pattern. Placing junk code at the beginning disrupts this natural tendency to focus on early information, interfering with the model's normal processing pattern. Another explanation is that placing junk code at the front pushes relevant code further into the less-effectively processed middle positions. This observation also suggests that the Transformer's attention mechanism prioritizes early tokens in the code sequence. When irrelevant information (junk code) occupies these critical early positions, it significantly impairs the model's ability to process the relevant code that follows.

2) The Opposite Result of GS-9 and GS-10: In Java, curly brackets can be eliminated if a compound block contains only one statement. The transformations *add / delete curly brackets (GS-9 and GS-10)* involve adding curly brackets to or removing them from a single-statement block. As observed from Table XI,

TABLE XI
RESULTS OF PARTIAL CODE TRANSFORMATIONS ON THE CODE COMPLETION TASK. THE RED AND GREEN COLORS INDICATE THE DEGREE OF DECREASE AND INCREASE, RESPECTIVELY

Java		MRR (code)			MRR (AST)		
Type	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
ID-2	abs.	0.7169	0.7069	↓ 1.40	0.7896	0.8050	↑ 1.95
	rel.	0.7255	0.7177	↓ 1.07	0.7916	0.8073	↑ 1.99
GS-9	abs.	0.7157	0.7264	↑ 1.48	0.7956	0.8018	↑ 0.77
	rel.	0.7259	0.7364	↑ 1.45	0.7973	0.8036	↑ 0.79
GS-10	abs.	0.7248	0.7121	↓ 1.75	0.7986	0.7919	↓ 0.84
	rel.	0.7333	0.7210	↓ 1.68	0.8007	0.7940	↓ 0.83
I-1	abs.	0.7169	0.6529	↓ 8.93	0.7896	0.7260	↓ 8.05
	rel.	0.7255	0.6592	↓ 9.14	0.7916	0.7274	↓ 8.11
I-2	abs.	0.7169	0.6442	↓ 10.15	0.7896	0.7141	↓ 9.56
	rel.	0.7255	0.6495	↓ 10.47	0.7916	0.7147	↓ 9.71
I-1(m)	abs.	0.7169	0.7129	↓ 0.56	0.7896	0.7851	↓ 0.57
	rel.	0.7255	0.7209	↓ 0.64	0.7916	0.7873	↓ 0.54
Python		**MRR (code)**			**MRR (AST)**		
Type	Pos.	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
ID-2	abs.	0.6597	0.6212	↓ 5.83	0.7571	0.7658	↑ 1.15
	rel.	0.6705	0.6301	↓ 6.03	0.7591	0.7667	↑ 1.01
I-1	abs.	0.6592	0.6272	↓ 4.85	0.7567	0.7255	↓ 4.12
	rel.	0.6701	0.6360	↓ 5.09	0.7587	0.7272	↓ 4.14
I-2	abs.	0.6597	0.5765	↓ 12.60	0.7571	0.6940	↓ 8.33
	rel.	0.6705	0.5913	↓ 11.82	0.7591	0.6941	↓ 8.56
I-1(m)	abs.	0.6585	0.6571	↓ 0.21	0.7562	0.7547	↓ 0.19
	rel.	0.6695	0.6679	↓ 0.23	0.7582	0.7567	↓ 0.19

TABLE XII
STATISTICS OF CONTROL STATEMENT

	Trainset	**Testset**
Control number	258,410	18,115
Block number	250,232	16,248
Single number	57,874 (10.79%)	4,958 (27.37%)
Predict probability	-	16.99%

these transformations exhibit opposite effects on both Transformer models during the code completion task. Specifically, adding curly brackets (GS-9) tends to enhance code completion performance, evidenced by a 1.48% increase in MRR. In contrast, removing curly brackets (GS-10) leads to a performance decline, with a 1.75% decrease in MRR. This pattern suggests that curly brackets provide clear structural cues that aid the model in predicting subsequent tokens, whereas their absence may introduce ambiguities or complexities in delineating code block boundaries, thus impeding accurate predictions.

Further analysis quantified the presence of curly brackets in single-statement blocks, as detailed in Table XII. The percentage of single-statement blocks is markedly higher in the test set (27.37%) compared to the training set (10.79%). Furthermore, the Transformer model exhibits a tendency to predict the curly bracket "{" following a control statement with a probability of 16.99%, indicating that it has learned this pattern from the coding style in the training data. The disparity in the occurrence rates of single statement blocks between the training and test sets (10.79% vs. 27.37%, respectively) may contribute to discrepancies in model performance and generalization capabilities across data sets. Combined with the performance changes mentioned above, these observations underscore that the recognition of patterns and structural signals, such as the presence of curly brackets, can enhance the ability of the Transformer model to parse and understand code, particularly in distinguishing between different blocks and scopes.

Finding 7: Transformer learns patterns from training data, recognizing structural signals like curly brackets, to enhance code understanding.

3) The Improvement of ID-2: Analyzing the impacts of the ID-2 transformation, which involves inserting junk code into existing code, we observed distinct effects on code completion tasks for seq-based and AST-based Transformers, as detailed in Table XI.

For the seq-based models, introducing junk code leads to a performance decrease in both Java and Python languages. For example, the MRR score for Java decreases by 1.40% and 1.07% at the absolute and relative positions, respectively. For Python, performance drops 5.83% and 6.03% in the same positions. This reduction suggests that the linear nature of seq-based Transformer makes it particularly sensitive to the noise introduced by junk code, impacting its ability to accurately predict the next token in code completion tasks. In contrast, the performance of the AST-based Transformer shows an improvement. For Java, there is an increase of 1.95% and 1.99% in MRR at absolute and relative positions, respectively. Similarly, for Python, the improvements are 1.15% and 1.01%. The resilience of the AST-based Transformer to junk code can likely be attributed to its structural parsing of code into trees. This structural approach may enable it to better isolate and disregard non-essential elements, providing a robust context for predicting code elements. Despite the introduction of irrelevant code snippets, the essential structural patterns remain unaffected, allowing the model to either maintain or enhance its predictive accuracy in code completion tasks. Additionally, the simplicity and basic nature of the inserted code probably contribute to its predictability at the AST level.

However, both seq-based and AST-based Transformers exhibit a lack of robustness in code summarization and code search tasks, as indicated in Tables IX and X where the default transformation of ID-2 is the middle position. For instance, the seq-based Transformer experiences a considerable decline in performance under junk code in the code search task, with the MRR score decreasing by 25.91% and 19.26% for Java and Python, respectively. Similarly, the AST-based Transformer demonstrates reductions of 9.81% for Java and 15.13% for Python. This suggests that although the AST-based Transformer model is more capable of managing semantic noise than seq-based Transformer, junk code still negatively impacts its ability to match code semantics with search queries, possibly due to the obfuscation of important syntactic features. In the code summarization task, the seq-based Transformer demonstrates a notable reduction in BLEU scores, with a drop of 12.06% in Java and 11.41% in Python, respectively. The performance of the AST-based Transformer also declines, though less drastically than its seq-based Transformer, with reductions of 8.54% and 5.28% for Java and Python. This implies that even a basic, non-functional code element can compromise the quality of the summarization by making it difficult for the model to understand the code's overall meaning or functionality.

Although the AST-based Transformers demonstrate a slight improvement in the code completion task when faced with junk code, their performance declines in both the code search and code summarization tasks. This reveals how junk code impacts model performance, which is heavily dependent on the specific task and the underlying model architecture.

4) Different Identifier Transformation: In the experimental results presented in Sections IV-A and IV-B, the impact of the *identifier transformation* on seq-based and AST-based Transformers is discussed. In this section, we focus on two included strategies: *function rename transformation (I-1)* and *variable rename transformation (I-2)*. Strategy I-1 renames all function and class names in the program, including both built-in and imported functions, while strategy I-2 targets only variable names, preserving the original function names.

Tables XI and XIII present the effects of I-1 and I-2 on the code completion and summarization tasks, respectively. Both I-1 and I-2 substantially affect Transformer performance across scenarios, including seq-based and AST-based models, various positional encodings, programming languages, and code intelligence tasks. For instance, I-1, which renames functions, reduces the MRR in Java's code completion task by 9.14% for seq-based and 8.11% for AST-based Transformers with relative positional encoding. I-2, which renames variables, decreases the BLEU scores in Python's code summarization by 8.77% for seq-based and 8.97% for AST-based Transformers with absolute positional encoding. This sensitivity indicates that Transformer-based code intelligence models heavily rely on identifier names to infer semantic meaning and make accurate predictions. Additionally, renaming variables (I-2) generally impacts Transformer performance more significantly than renaming functions (I-1). under I-2, the seq-based Transformer with absolute positional encoding shows MRR decreases of 10.15% for Java and 12.60% for Python in code completion tasks, whereas under I-1, the reductions are 8.93% and 4.85%, respectively.

To further investigate the significance of identifiers in enabling code intelligence models to understand the purpose and flow of the code, we conduct a new study focusing on the impact of the *function rename transformation (I-1)* by specifically targeting the main function of the input code, denoted as I-1(m). This approach only renames the main function name while preserving all other identifiers unchanged. For instance, in the code snippet shown in Fig. 1, only the main function name `createTypesTableNames` would be renamed under I-1(m). However, when we examine the impact of I-1(m) on the generation of code summaries, as shown in Table XIII, we observe a notable effect on both seq-based and AST-based Transformers. The influence on the performance of the Transformer model was close to 2%. Considering that we do not

TABLE XIII
RESULTS OF IDENTIFIER TRANSFORMATIONS ON THE CODE SUMMARIZATION TASK. THE RED COLOR INDICATES THE DEGREE OF DECREASE

Java Type	Pos.	BLEU (code) w/o. t.	BLEU (code) w. t.	BLEU (code) imp. (%)	BLEU (AST) w/o. t.	BLEU (AST) w. t.	BLEU (AST) imp. (%)
I-1	abs.	13.80	12.79	↓ 7.27	13.94	13.15	↓ 5.64
	rel.	13.70	12.81	↓ 6.54	13.82	12.98	↓ 6.10
I-2	abs.	13.80	12.59	↓ 8.77	13.94	12.69	↓ 8.97
	rel.	13.70	12.92	↓ 5.72	13.82	13.14	↓ 4.97
I-1(m)	abs.	13.80	13.59	↓ 1.52	13.94	13.68	↓ 1.87
	rel.	13.70	13.45	↓ 1.80	13.82	13.58	↓ 1.78
Python Type	**Pos.**	**w/o. t.**	**w. t.**	**imp. (%)**	**w/o. t.**	**w. t.**	**imp. (%)**
I-1	abs.	13.82	13.33	↓ 3.55	12.93	12.77	↓ 1.24
	rel.	14.32	13.88	↓ 3.03	13.79	13.46	↓ 2.39
I-2	abs.	13.85	13.00	↓ 6.11	12.93	12.79	↓ 1.08
	rel.	14.35	13.76	↓ 4.11	13.81	13.44	↓ 2.73
I-1(m)	abs.	13.50	13.22	↓ 2.10	12.76	12.67	↓ 0.68
	rel.	13.98	13.73	↓ 1.81	13.52	13.34	↓ 1.36

TABLE XIV
CODE SUMMARY OF THE EXAMPLE CODE IN FIG. 1 UNDER DIFFERENT CODE TRANSFORMATIONS

No.	Code summary
ID-1	creates a new ; instance .
ID-2	creates a new table descriptor .
ID-4	imports a texture file from a file .
I-1	return the names of the array of strings .
I-2	create the types of the types .
Original prediction: create the table names for the given table.	

split subtokens for code, meaning that a single token change, specifically the renaming of the main function, can significantly reduce Transformer performance in code summarization tasks. This experiment highlights the importance of the function name for Transformers in generating accurate code summaries, especially for the main function name. The main function name likely serves as a crucial point for the Transformer to understand the overall purpose and structure of the code, influencing its ability to generate high-quality summaries.

Based on the above experimental results, we can infer that identifiers, including both function names and variable names, play a vital role in Transformer's ability to perform well in various code intelligence tasks.

5) Case Study of Code Transformation: In this section, we conduct a case study to analyze the greatest impact of the types of code transformation, *insertion / deletion transformation* and *identifier transformation* on generating code summary. Table XIV presents the code summaries generated for the example code (seen in Fig. 1(a)) under different code transformation strategies.

Given that the example code supports only the transformations of ID-1, ID-2, and ID-4, we present summaries generated solely from these transformed codes. Our observations indicate that the Transformer misinterpreted the transformed programs for ID-1, ID-2, and ID-4, resulting in inaccurate descriptions. Additionally, the code summaries for ID-1 and ID-4 include terms not found in the original program, originating from the noisy segments inserted. This outcome indicates that the Transformer's performance is markedly compromised by the inability to recognize the inserted code segments. Moreover, the Transformer demonstrates poor performance in generating code summaries during *identifier transformation*. Despite all terms being from the original program, the Transformer fails to comprehend the input code in the summary generation process. These findings underscore the critical role of identifiers in enabling the Transformer to generate precise code summaries.

In summary, this case study highlights the varied impacts of *insertion / deletion transformation* and *identifier transformation* on the Transformer's ability to generate code summaries. It reveals the Transformer's vulnerability to the irrelevant inserted code segments and underscores the critical importance of identifiers in achieving precise code summarization.

TABLE XV
THE AVERAGE EDIT DISTANCE OF FIVE TYPES OF CODE TRANSFORMATION

Type	Language	Distance
Block transformation	java	12.23
	py	16.63
Insertion / deletion transformation	java	30.21
	py	15.57
Grammatical statement transformation	java	6.56
	py	7.86
Grammatical token transformation	java	3.20
	py	3.04
Identifier transformation	java	16.62
	py	13.50

6) Impact of Edit Distance: To quantify the degree of code transformation, we utilize the edit distance metric, which calculates the minimum number of operations required to transform the original code into the transformed code. Table XV illustrates the edit distances between the original and transformed code for each transformation type.

We observe that *insertion/deletion transformation* has the highest edit distance, reflecting substantial changes in the transformed code. This suggests that the Transformer's ability to generate accurate predictions may become more challenging as the amount of code changes increases. Moreover, *identifier transformation* and *block transformation* exhibit a comparable level of edit distance, but they have different impacts on the robustness of the Transformer. *Identifier transformation* has a more substantial impact on the Transformer's performance, implying that the content of the code changes, such as changes in identifiers, is more critical for the Transformer. On the other hand, T_{GS} and T_{GT} have minimal edit distances and a negligible impact on the Transformer's performance, implying that these transformation types are less likely to affect the model's performance.

Table XVI presents the results of the correlation analysis between the impact of performance and the edit distance at

TABLE XVI
THE CORRELATION ANALYSIS BETWEEN PERFORMANCE IMPACT AND EDIT DISTANCE IN STRATEGY LEVEL

Lang.	Type	Pos.	CC	CSM			CS
			MRR	bleu	rog	meteor	MRR
java	code	abs	-0.5184	-0.6898	-0.7361	-0.4553	-0.5092
		rel	-0.5476	-0.7937	-0.8022	-0.7380	-0.4950
	AST	abs	-0.4176	-0.6365	-0.6520	-0.6997	-0.0991
		rel	-0.4182	-0.7574	-0.7896	-0.7860	-0.0739
python	code	abs	-0.1462	-0.4297	-0.7096	-0.2022	-0.0956
		rel	-0.0995	-0.4941	-0.8112	-0.4559	-0.0870
	AST	abs	-0.4043	-0.2480	-0.3803	-0.0928	-0.0845
		rel	-0.4468	-0.3861	-0.1763	-0.4512	-0.0907

the strategic level. The negative correlation results suggest that there is no positive correlation between the degree of code transformation and the impact on model performance. Moreover, there are notable differences among code intelligence tasks. For example, in the code completion task, most model results hover around -0.5, indicating a low negative correlation. In contrast, in the code summarization task, the correlation of certain model results approaches -0.8, signifying a more pronounced negative correlation. In the code search task, the majority of the model results are below -0.1 and the correlation analysis does not reveal a significant correlation.

Finding 8: The impact of code transformation on model performance does not increase along with the increase of change degree of strategy.

D. Answer to RQ4: Impact of Additional Factors

In this section, we explore additional factors that might mitigate the effects of code transformation on the robustness of the Transformer. Specifically, we investigate various approaches developed for traversing the AST's tree structure within the Transformer in Section IV-D1. Subsequently, we examine whether variations in token embedding patterns can enhance the Transformer's robustness in Section IV-D2. Finally, we assess the impact of model parameters on the robustness of the Transformer in Section IV-D3.

1) Impact on Tree-Based Transformer: In order to investigate the robustness of an AST-based Transformer under code transformation, we directly traverse AST (depth-first-search order) into a sequence and apply two popular positional encoding strategies (absolute positional encoding and relative positional encoding) in Section IV-B. However, this traversal strategy may lead to a loss of structural information in the AST. For further analysis, we select two approaches developed specifically for utilizing tree structure in Transformer [8], referred to as Tree-based Transformer. Tree-based Transformers operate directly on the tree structures of ASTs without linearizing them into sequences, thereby preserving more structural information.

The positional encoding of sequences is based on the distance between two positions, whereas in trees, the relationship between two nodes is determined by the steps taken along the branches of the tree. As a result, traditional positional encoding techniques for sequences may not be suitable for ASTs. To address this issue, Shiv et al. [35] proposed tree positional encodings, named as **tree absolute positional encoding** in our study, to better encode tree structures. Moreover, Kim et al. [12] explored the use of Transformers on the tree structure of code and proposed tree relative attention based on the tree traversal order. Similar to sequential relative attention, the relative distance between two nodes can be defined as the shortest path between them, which we refer to as **tree relative positional encoding** in our experiments.

TABLE XVII
OVERALL RESULTS OF CODE TRANSFORMATION ON 2 TYPES OF TREE-BASED TRANSFORMER IN THE CODE COMPLETION TASK. THE "ABS." AND "REL." REPRESENT THE TREE ABSOLUTE POSITIONAL ENCODING AND TREE RELATIVE POSITIONAL ENCODING, RESPECTIVELY

Type	Tree.	MRR (Java)			MRR (Python)		
		w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
T_{B}	abs.	0.6949	0.6993	↑0.64	0.7650	0.7644	↓0.07
	rel.	0.8069	0.8089	↑0.26	0.7669	0.7666	↓0.04
T_{ID}	abs.	0.6928	0.6691	↓3.42	0.7589	0.7364	↓2.97
	rel.	0.7910	0.7544	↓4.63	0.7605	0.7398	↓2.73
T_{GS}	abs.	0.6292	0.6296	↑0.07	0.7581	0.7528	↓0.70
	rel.	0.8042	0.8013	↓0.36	0.7597	0.7548	↓0.64
T_{GT}	abs.	0.6379	0.6311	↓1.06	0.7553	0.7482	↓0.94
	rel.	0.8029	0.7979	↓0.62	0.7575	0.7514	↓0.81
T_{I}	abs.	0.6398	0.6144	↓3.98	0.7592	0.7165	↓5.62
	rel.	0.7930	0.7230	↓8.83	0.7606	0.7125	↓6.33
T_{all}	abs.	0.6589	0.6487	↓1.55	0.7593	0.7437	↓2.06
	rel.	0.7996	0.7771	↓2.81	0.7610	0.7450	↓2.11

Table XVII presents the overall results of the impact of code transformations on the tree-based Transformer in the code completion task. A comparison between Tables VI and XVII reveals that utilizing the tree structure information in the ASTs enables the Transformer to maintain a more robust performance than the model that directly traverses the ASTs in a sequence. For example, the overall MRR score of the tree-based Transformer using tree absolute positional encoding has a decrease of 1.55% in the Java's code completion task, compared to a 2.88% reduction in the AST-based Transformer (seen in Table VI). For Python, the performance changes are 2.06% vs. 2.25%, respectively. In scenarios using relative positional encoding, the performance decrease of the tree-based Transformer closely matches that of the AST-based Transformer in both Java (2.81% vs. 2.60%) and Python (2.11% vs. 2.12%). These results demonstrate that tree absolute positional encoding is an effective approach to improve the robustness of Transformer.

Notably, the tree-based Transformer employing absolute positional encoding demonstrates more robustness than those using relative positional encoding under code transformations, which is different from seq-based and AST-based Transformer. Specifically, the Transformer with tree absolute positional encoding outperforms all other models under the *Identifier transformation* transformation, with a decrease of 3.98% in Java and 5.62% in Python. As discussed previously, the Transformer

TABLE XVIII
RESULTS OF CODE TRANSFORMATION ON THE PERFORMANCE OF TRANSFORMER FOR THE CODE SUMMARIZATION TASK UNDER SUB-TOKEN

Java		BLEU			ROUGE-L			METEOR		
Type	Pos.	w/o. t.	w. t	Imp.(%)	w/o. t.	w. t	Imp.(%)	w/o. t.	w. t	Imp.(%)
T_{B}	abs.	12.88	12.88	↓0.04	25.80	25.77	↓0.09	8.92	8.89	↓0.33
	rel.	15.24	15.25	↑0.08	31.44	31.49	↑0.15	11.33	11.35	↑0.17
T_{ID}	abs.	14.41	13.66	↓5.18	27.03	25.17	↓6.89	9.53	8.53	↓10.44
	rel.	16.50	14.89	↓9.78	32.23	28.12	↓12.77	11.88	9.94	↓16.31
T_{GS}	abs.	13.93	13.90	↓0.25	27.48	27.49	↑0.03	10.04	10.04	↓0.09
	rel.	16.56	16.71	↑0.95	32.62	32.87	↑0.75	12.45	12.52	↑0.53
T_{GT}	abs.	13.85	13.85	↓0.03	26.85	26.60	↓0.91	9.67	9.51	↓1.61
	rel.	16.15	16.12	↓0.17	32.43	32.44	↑0.03	12.01	12.01	↓0.03
T_{I}	abs.	14.98	13.45	↓10.18	29.01	24.91	↓14.11	10.54	8.45	↓19.85
	rel.	17.02	14.89	↓12.53	34.01	28.43	↓16.41	12.76	10.04	↓21.32
T_{all}	abs.	14.01	13.55	↓3.31	27.23	25.99	↓4.56	9.74	9.08	↓6.74
	rel.	16.29	15.57	↓4.43	32.55	30.67	↓5.77	12.09	11.17	↓7.57
Python		BLEU			ROUGE-L			METEOR		
Type	Pos.	w/o. t.	w. t	Imp.	w/o. t.	w. t	Imp.	w/o. t.	w. t	Imp.
T_{B}	abs.	14.27	14.24	↓0.19	26.01	25.83	↓0.71	8.93	8.86	↓0.83
	rel.	15.95	15.94	↓0.01	30.64	30.64	0.00%	11.83	11.83	↓0.03
T_{ID}	abs.	14.99	14.55	↓2.96	27.11	25.66	↓5.33	9.47	8.73	↓7.82
	rel.	16.76	15.11	↓9.85	31.12	25.67	↓17.52	12.04	9.63	↓20.01
T_{GS}	abs.	14.25	14.19	↓0.40	26.29	26.17	↓0.48	9.13	9.01	↓1.26
	rel.	16.12	16.08	↓0.23	30.93	30.95	↑0.04	12.05	12.00	↓0.44
T_{GT}	abs.	14.66	14.54	↓0.83	26.24	25.89	↓1.33	8.85	8.73	↓1.36
	rel.	16.29	16.23	↓0.36	30.54	30.10	↓1.43	11.75	11.47	↓2.40
T_{I}	abs.	15.07	14.20	↓5.82	28.02	25.38	↓9.40	9.95	8.74	↓12.18
	rel.	16.77	14.86	↓11.39	31.96	25.86	↓19.10	12.45	9.65	↓22.44
T_{all}	abs.	14.65	14.34	↓2.08	26.73	25.79	↓3.54	9.26	8.81	↓4.88
	rel.	16.38	15.64	↓4.47	31.04	28.64	↓7.72	12.02	10.92	↓9.21

model often struggles to adapt to changes in variable and function names, which are crucial to understanding the program semantics. This suggests that Transformer with tree absolute positional encoding is particularly effective in handling identifier transformations, and maintains an understanding of the code's structure and semantics despite changes in the identifiers themselves.

Finding 9: More effective approaches to learning the code representations based on ASTs can improve the robustness of Transformer.

2) Impact of Sub-Token: In our previous experiments, we did not split tokens into sub-tokens to ensure a one-to-one correspondence between the AST nodes and their respective values, as discussed in Section III-B1. However, some studies have shown that utilizing sub-tokens or byte-pair encoding can enhance the performance of code intelligence tasks [14], [50]. To explore whether a sub-token vocabulary can increase the robustness of the Transformer model, we analyze its effect under code transformations in the code summarization task in this section. During data preprocessing, we specifically split the source code into sub-tokens if they are formatted in `CamelCase` or `snake_case`. Then, we conduct experiments with sub-tokenization in the code summarization task using the same code transformation strategies and model configurations as before.

Table XVIII presents the overall results of the seq-based Transformer using sub-tokens in the code summarization task. Compared to the results without sub-tokenization (as shown in Table V), employing sub-tokens markedly improves all evaluative scores for code summarization, indicating enhanced processing capabilities of the Transformer. However, despite these improvements in summarization quality, sub-tokenization does not enhance the Transformer's robustness under code transformation. Transformer using sub-tokens performs better than without sub-tokenization in Python code summarization, but worse in Java code summarization. When using sub-tokens, the Transformer's BLEU, ROUGE-L, and METEOR scores decrease by 2.08%, 3.54%, and 4.88% for Python, and 3.31%, 4.56%, and 6.74% for Java. In contrast, the declines without sub-tokenization are 2.75%, 4.01%, and 9.67% for Python, and 2.94%, 3.96%, and 8.96% for Java. This indicates that while sub-tokenization improves overall performance, it does not necessarily provide additional resilience against code transformation's disruptive effects.

Finding 10: Sub-token improves the performance of Transformer, but does not increase the model's robustness against code transformation.

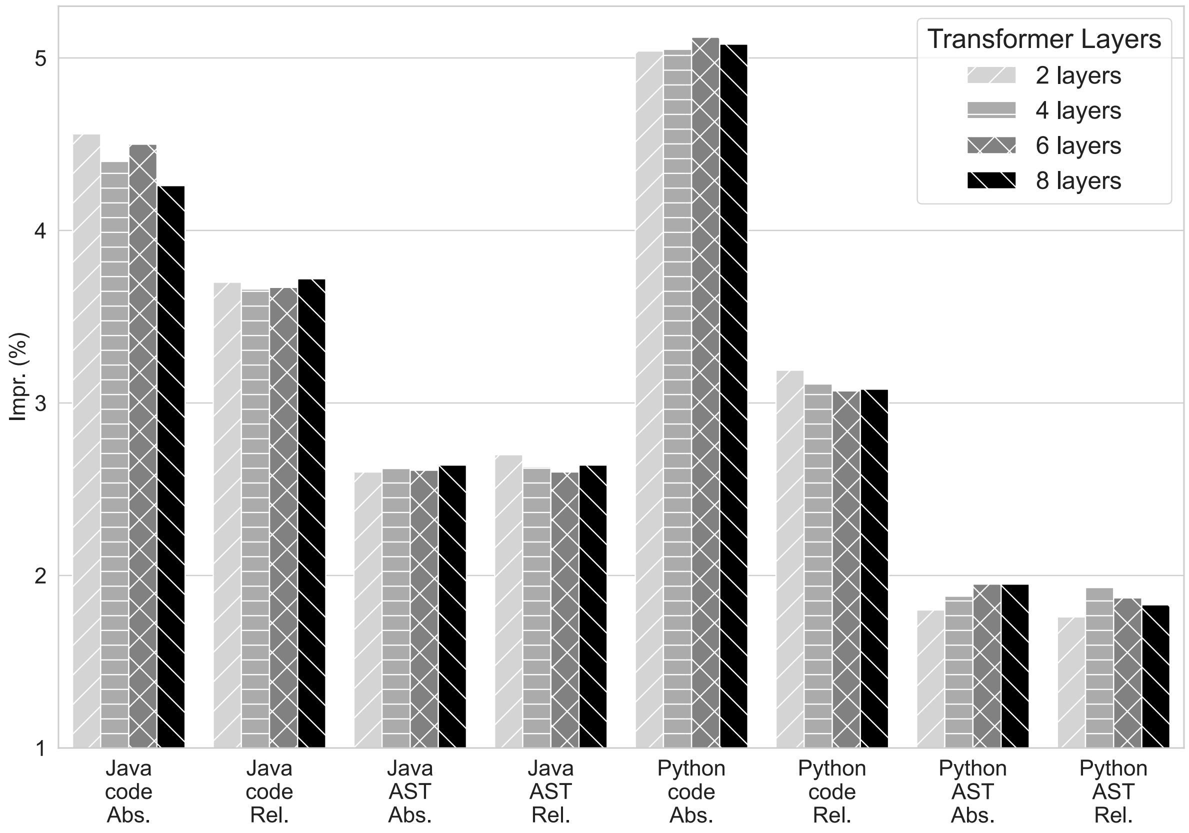


Fig. 3. Effect of Transformer layer on different programming languages, data types and encoding strategies.

3) Impact of the Model Parameter: In our previous experiments, we established the Transformer with the layer at $l = 6$ across various code intelligence tasks. This section delves into the influence of altering the Transformer layer on the robustness under code transformation, especially in the code completion task.

Fig. 3 illustrates the performance changes under code transformation for diverse Transformer model configurations. Each colored bar signifies a different layer count, with the x-axis categorizing the Transformer models by programming languages, data types, and positional encoding strategies. The y-axis quantifies performance changes using Mean Reciprocal Rank (MRR) metrics. Observations reveal that MRR metrics exhibit relative stability with increasing layer number, suggesting that increasing the number of layers does not inherently enhance robust performance.

To determine the statistical significance of layer count variations on robustness, we conducted a Friedman test, a non-parametric analog to repeated measures ANOVA, applicable when data defy the prerequisites of parametric tests, such as normal distribution of residuals or variance homogeneity. The Friedman test yielded a statistic of approximately 0.65 and a p-value of roughly 0.89. Given the p-value exceeds the standard alpha threshold of 0.05, the null hypothesis, that there is no significant difference in performance across various layer counts, stands unrefuted. Consequently, our analysis does not support any statistically significant divergence in the robustness of the transformer under varying transformer layers.

Moreover, our findings reinforce the earlier observations with a 6-layer Transformer configuration, demonstrating consistency even as the number of layers varies. For instance, the AST-based Transformer model consistently outperforms the sequence-based Transformer model in robustness, independent of layer count. Similarly, we noted that positional encoding strategies exert a more pronounced effect on sequence-based Transformers compared to AST-based models.

Finding 11: Increasing the number of Transfomer layers does not contribute to the robustness under code transformation.

E. Answer to RQ5: Extension of Code Transformation

In this section, our aim is to explore additional application scenarios of code transformations to further evaluate their potential. In addition, we sought to verify the findings observed in the Transformer model. We fine-tune pre-trained models and apply our code transformations to assess robustness in Section IV-E1. Moreover, we examine another Transformer-based code intelligence task, code translation, to further investigate its robustness in Section IV-E2.

1) Impact on Pre-Training Models: In this section, we fine-tuned the pre-training models in our tasks and evaluated

TABLE XIX
RESULTS OF PRETRAIN MODEL (CODEGPT) ON THE CODE COMPLETION TASK

CodeGPT	MRR (Java)			MRR (Python)		
Type	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
T_B	0.6711	0.6688	↓0.35	0.6362	0.6345	↓0.27
T_{ID}	0.6539	0.6414	↓1.92	0.6468	0.6135	↓5.15
T_{GS}	0.6788	0.6741	↓0.70	0.6303	0.6252	↓0.81
T_{GT}	0.6701	0.6647	↓0.80	0.6403	0.6338	↓1.02
T_I	0.6625	0.6236	↓5.87	0.6543	0.6130	↓6.32
T_{all}	0.6673	0.6545	↓1.92	0.6416	0.6249	↓2.74

TABLE XX
RESULTS OF PRETRAIN MODEL OF CODEBERT ON THE CODE SUMMARIZATION TASK

CodeBert	BLEU (Java)			BLEU (Python)		
Type	w/o. t.	w. t.	imp. (%)	w/o. t.	w. t.	imp. (%)
T_B	14.72	14.71	↓0.06	17.64	17.55	↓0.50
T_{ID}	15.19	12.87	↓15.28	18.91	17.04	↓9.91
T_{GS}	16.16	15.90	↓1.61	17.73	17.69	↓0.23
T_{GT}	15.21	14.98	↓1.49	18.03	17.94	↓0.55
T_I	16.49	12.36	↓25.08	18.81	15.92	↓15.37
T_{all}	15.55	14.16	↓8.94	18.22	17.22	↓5.48

the impact of code transformations on pre-trained models. Our objective is to determine whether pre-training models can maintain robustness under code transformation.

For code summarization, we fine-tuned the model using the CodeSearchNet dataset, and follows the according to implementation by Microsoft[5]. We reproduce the experiments with the default hyperparameters setting and evaluation metric (BLEU). Considering that CodeBERT is not suitable for a masked language task, we use another model, CodeGPT, which is a pre-trained language model designed for programming language, for the code completion task following the work of Kim et al. [12]. We reproduce the experiments with the default hyper-parameters setting from CodeXGLUE[6].

Tables XIX and XX present the overall results for the code completion and code summarization tasks using pre-trained models, respectively. In comparison with the seq-based Transformer, CodeGPT demonstrates greater robustness in the code completion task. For instance, the overall MRR score of CodeGPT exhibits a decrease of only 1.92% in predicting Java code, as indicated in Table XIX, while the seq-based Transformer shows a more significant decrease of 5.42% for (referenced in Table III). Similar patterns are observed in Python, where the MRR for CodeGPT decreases by 2.74%, compared to a 6.16% decrease for the Transformer. These results suggest that pre-trained models such as CodeGPT are more robust in code prediction than traditional Transformer models.

However, CodeBert performs worse than the seq-based Transformer under code transformation. In the code summarization task, the overall performance decreases in the seq-based Transformer are 2.94% and 2.75% (see Table V) for Java and Python, while the decreases in CodeBert are 8.94% and 5.48% (see Table XX), respectively. More specifically, the *identifier transformation* results in a significant drop in CodeBert performance, which is substantially greater than Transformer. The seq-based Transformer has decreased BLEU by 8.02% and 4.83% for Java and Python, respectively, while CodeBert has decreased BLEU by 25.08% and 15.37%. This suggests that CodeBERT relies more heavily on identifiers when generating code summaries, making it more vulnerable to identifier transformations compared to the Transformer.

Finding 12: CodeGPT is more robust to code transformations in the code completion task than Transformer, whereas CodeBERT exhibits greater vulnerability, particularly to identifier transformations in code summarization tasks.

2) Impact on Code Translation Task: To investigate the effects of semantic-preserving code transformations on code translation tasks, we utilize TransCoder, a Transformer-based code translation approach supporting Java and Python languages [69]. Our experiments leverage the CodeGen benchmark[7] to assess how code transformation influences code translation performance. The objective of code translation—converting code from one programming language to another—is a pivotal task in programming language research [70], [71], [72]. Using TransCoder as our baseline, our objective is to quantify the impact of code transformation on translation quality.

The CodeGen benchmark facilitates evaluation using GeeksForGeeks scripts[8]. Each script corresponds to a specific function and programming language, containing a reference function (`f_gold`), a placeholder (marked as `TOFILL`) for the generated function, and a *main* function. This format helps to determine the viability of the translated code. The main function includes ten test cases to execute and evaluate both the reference and generated functions, comparing their behaviors across four categories: success, failure, error, and timeout. We adopted the success rate, defined as the proportion of successful translations over total behaviors, as a metric for successful translation, denoted as CA@1, following [69].

Furthermore, we adhere to the evaluation criteria outlined in Section III-C2, comparing only the transformable code for each transformation strategy. Strategies yielding fewer than ten transformable code instances were excluded from consideration to avoid randomness. Additionally, a CA@1 score of zero for transformable code of any strategy before applying code transformations was disregarded, ensuring a focus on viable transformation strategies that demonstrate measurable impact on code translation outcomes.

[5]https://github.com/microsoft/CodeBERT
[6]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/CodeCompletion-token
[7]https://github.com/facebookresearch/CodeGen
[8]https://www.geeksforgeeks.org/

Table XXI presents TransCoder's performance impact under code transformation, following the experimental setup outlined in [69], with results provided for beam sizes 1 and 10. From the table, we can observe interesting distinct results, contrasting prior discussions on code completion, code summarization, and code search.

In the **block transformation**, disparate outcomes emerge between the Java-to-Python and Python-to-Java code translation tasks. Notably, in the Python-to-Java task, block transformations predominantly result in an improvement in CA@1, indicating an enhancement in the quality of generated code. Conversely, in the Java-to-Python task, CA@1 values mostly exhibit a decline. This divergence is attributed to linguistic disparities between the two programming languages. Particularly noteworthy is the impact of the B-7 transformation, "extract function", which transforms portions of code into new internal functions. This transformation notably results in a decrease in CA@1 from 0.279 to 0 for 323 samples, in contrast to its minimal effects in other tasks. We posit that this discrepancy is intricately tied to the nature of the evaluation data. As previously expounded, the structured format of GeeksForGeeks scripts, wherein a main function comprises 10 test cases to validate the equivalence between reference and translated functions, poses challenges when new functions are introduced, significantly impairing performance.

In the **insertion / deletion transformation**, substantial disparities arise due to language-specific idiosyncrasies, particularly evident in the varying impact of transformations across Java-to-Python and Python-to-Java tasks. For instance, the ID-2 transformation, "insert junk code", exhibits limited effects in the Java-to-Python task but substantially impacts CA@1 in the Python-to-Java task, resulting in a decline from 0.315 to 0.019 (beam size = 1) and 0.016 (beam size = 10). Furthermore, the impact of the ID-4 transformation, "import unrelated library", is more pronounced in the Python-to-Java task than in its counterpart. This disparity is attributed to the more stringent syntactic structure of Java code, facilitating easier identification of pertinent code segments in code translation tasks, while the flexibility inherent in Python poses challenges for the model to discern crucial code amidst noise.

In the **grammatical statement transformation**, the most substantial impact is observed in the GS-1 transformation, "refactor return statement". This transformation will declare a variable and return it if the return-statement returns an integer literal. After checking the translated code, we find that the omission of return variable declaration in Python leads to syntax errors upon translation to Java. This underscores inherent limitations in cross-language code translation models, particularly concerning variable initialization, resulting in erroneous code translation.

The **grammatical token transformation**, exclusive to Python-to-Java translation, exhibits significant discrepancies across the GT-1, GT-3, and GT-4 transformation strategies, in contrast to their negligible effects in prior tasks. These disparities predominantly stem from the distinctive language features of Java and Python, compounded by the inherent

TABLE XXI
RESULTS OF CODE TRANSFORMATIONS ON THE CODE TRANSLATION TASK. COLUMNS REPRESENT THE TRANSFORMATION STRATEGY'S ID, STRATEGY NAME, NUMBER OF TRANSFORMABLE INSTANCES FOR THIS STRATEGY, AND THE CHANGES OBSERVED BEFORE AND AFTER APPLYING THE TRANSFORMATION, RESPECTIVELY. EACH TRANSFORMATION STRATEGY IS REPRESENTED BY TWO ROWS; THE UPPER ROW DISPLAYS RESULTS WITH A BEAM SIZE OF 1, WHILE THE LOWER ROW SHOWS RESULTS WITH A BEAM SIZE OF 10. THE SYMBOL "-" REPRESENTS THE ABSENCE OF TRANSFORMABLE CODE FOR THIS STRATEGY

No	Trans.	Java → Python		Python → Java	
		N	Changes	N	Changes
B-1	For to While	386	0.409 → 0.376 ↓	341	0.282 → 0.343 ↑
			0.409 → 0.412 ↑		0.282 → 0.358 ↑
B-2	While to For	112	0.429 → 0.098 ↓	-	-
			0.429 → 0.098 ↓	-	-
B-3	Elseif to Else If	48	0.417 → 0.417	41	0.122 → 0.171 ↑
			0.417 → 0.375 ↓		0.122 → 0.146 ↑
B-4	Else If to Elseif	48	0.417 → 0.417	8	0.0 → 0.0
			0.417 → 0.375 ↓		0.0 → 0.125 ↑
B-5	If-Else Swap	152	0.401 → 0.322 ↓	136	0.147 → 0.25 ↑
			0.401 → 0.336 ↓		0.147 → 0.294 ↑
B-6	Decompose Complex If	86	0.488 → 0.512 ↑	83	0.205 → 0.253 ↑
			0.488 → 0.558 ↑		0.205 → 0.217 ↑
B-7	Extract Function	-	-	323	0.279 → 0.0 ↓
		-	-		0.279 → 0.0 ↓
ID-1	Insert Comments	-	-	-	-
		-	-	-	-
ID-2	Insert Junk Code	616	0.49 → 0.404 ↓	616	0.315 → 0.019 ↓
			0.49 → 0.42 ↓		0.315 → 0.016 ↓
ID-3	Append Return Statement	-	-	117	0.111 → 0.111
		-	-		0.111 → 0.12 ↑
ID-4	Import Unrelated Library	616	0.49 → 0.44 ↓	616	0.315 → 0.06 ↓
			0.49 → 0.474 ↓		0.315 → 0.138 ↓
ID-5	Remove Comments	-	-	-	-
		-	-	-	-
ID-6	Replace Print with Pass	12	0.333 → 0.167 ↓	13	0.077 → 0.0 ↓
			0.333 → 0.25 ↓		0.077 → 0.0 ↓
ID-7	Delete Unused Variable	52	0.115 → 0.115	7	0.0 → 0.0
			0.115 → 0.115		0.0 → 0.0
GS-1	Refactor Return Statement	117	0.598 → 0.444 ↓	116	0.207 → 0.034 ↓
			0.598 → 0.53 ↓		0.207 → 0.034 ↓
GS-2	Internalize For Loop Declaration	17	0.353 → 0.353	-	-
			0.353 → 0.412 ↑	-	-
GS-3	Externalize For Loop Declaration	343	0.423 → 0.324 ↓	-	-
			0.423 → 0.37 ↓	-	-
GS-4	Separate Declaration&Initialization	443	0.44 → 0.393 ↓	-	-
			0.44 → 0.429 ↓	-	-
GS-5	Wrap with Logical NOT	545	0.468 → 0.182 ↓	444	0.23 → 0.205 ↓
			0.468 → 0.189 ↓		0.23 → 0.207 ↓
GS-6	Reverse Comparison Operator	497	0.459 → 0.3 ↓	285	0.228 → 0.228
			0.459 → 0.352 ↓		0.228 → 0.235 ↑
GS-7	Explicitize Assignment Operator	156	0.429 → 0.429	222	0.243 → 0.261 ↑
			0.429 → 0.429		0.243 → 0.275 ↑
GS-8	Expand Unary Operator	419	0.418 → 0.363 ↓	-	-
			0.418 → 0.391 ↓	-	-
GS-9	Encapsulate in Curly Braces	416	0.471 → 0.474 ↑	-	-
			0.471 → 0.498 ↑	-	-
GS-10	Remove Redundant Braces	197	0.391 → 0.376 ↓	-	-
			0.391 → 0.391	-	-
GT-1	Boolean to Integer	-	-	82	0.244 → 0.049 ↓
		-	-		0.244 → 0.037 ↓
GT-2	Integer to Boolean	-	-	1	1.0 → 0.0 ↓
		-	-		1.0 → 0.0 ↓
GT-3	Promote Integral Type	570	0.479 → 0.470 ↓	285	0.354 → 0.211 ↓
		570	0.479 → 0.493 ↑		0.354 → 0.214 ↓
GT-4	Promote Floating Type	586	0.486 → 0.469 ↓	285	0.354 → 0.0 ↓
		586	0.486 → 0.498 ↑		0.354 → 0.0 ↓
GT-5	Refactor Input API	-	-	1	0.0 → 1.0 ↑
		-	-		0.0 → 1.0 ↑
GT-6	Refactor Ouput API	11	0.364 → 0.364	13	0.077 → 0.0 ↓
		11	0.364 → 0.364		0.077 → 0.077
I-1	Rename Function and Class	616	0.49 → 0.506 ↑	616	0.315 → 0.308 ↓
			0.49 → 0.523 ↑		0.315 → 0.343 ↑
I-2	Rename Variable	616	0.49 → 0.362 ↓	616	0.315 → 0.304 ↓
			0.49 → 0.403 ↓		0.315 → 0.326 ↑

characteristics of GeeksForGeeks scripts. Java's strict data type definitions contrast with Python's flexibility, leading to misinterpretations upon translation, particularly impactful in scripts involving mathematical computations, wherein alterations in data types disrupt test case validation, consequently affecting CA@1 scores.

In the **identifier transformation**, the I-1 transformation, "rename function and class", can significantly impact the evaluation process of code translation. In our experiments, whether translating Python to Java or Java to Python, the original function names for all reference functions were labeled as `f_gold`, and the main function only compares the behavior between `f_gold` and `f_filled`. After translation, over 90% of the function names remained as `f_gold` in the translated code. To ensure a more accurate assessment of code translation quality, especially in terms of function functionality, we decided to standardize the function names in the translated code to `f_filled`. This adjustment aims to facilitate better evaluation of code translation, focusing on the core functionality rather than script evaluation. However, this modification will affect the measurement of I-1, as demonstrated in our previous tasks such as code search and code summarization, where function names did indeed influence the results due to the nature of the tasks.

On the other hand, the impact of I-2 (rename variable transformation) on code translation was not as significant compared to its impact on other tasks like code completion, summarization, and search. We attribute this to two main factors. Firstly, the code translation model tends to prioritize underlying semantic information, particularly evident in mathematical calculation functions within the dataset. Secondly, the identifiers in code scripts from GeeksForGeeks often consist of simple variables such as "a", "b", "c", "m", and "n", which are inherently intuitive. In contrast, the identifiers CodeSearchNet, consisting of real-world code repositories, often exhibit a wide range of personal styles and functionality-driven naming conventions. Consequently, the sensitivity of tasks such as code completion, summarization, and search to identifiers differs from that of code translation.

Indeed, the code translation model exhibits the lack of robustness under code transformation, but this task presents unique phenomena compared to the previous three tasks, primarily due to three reasons:

- Dataset Characteristics: The GeeksForGeeks scripts dataset differs significantly from CodeSearchNet, with a smaller sample size of only 616. This variance in dataset characteristics can influence model training, evaluation, and ultimately, the observed phenomena in code translation.
- Task Type Differences: Each type of code intelligence task has its own focus, leading to differences in data selection, training methods, and evaluation criteria. These variations in task types contribute to distinct phenomena observed in code translation compared to other tasks.
- Language Differences: Programming languages have distinct characteristics, syntax, and paradigms. Consequently, code translated from one language to another may not strictly adhere to the grammatical specifications of the target language, resulting in challenges. This language disparity is a crucial issue identified in code transformation for code translation tasks.

In light of these observations, it's essential for future code translation models to delve deeper into the paradigms of different programming languages. By leveraging the unique characteristics and structures of individual languages, models can optimize the generated code and achieve better cross-language translation effects. This approach would enable more accurate and contextually appropriate translations, enhancing the overall quality of code translation outputs.

V. Discussion

In this section, we will discuss from various perspectives based on the experimental results in Section IV, to explore the challenges and opportunities of Transformer-based code intelligence models, especially in robustness.

A. Do ASTs Really Make Transformers More Robust?

Abstract Syntax Trees (ASTs) represent the abstract syntactic structure of source code, and prior studies [35], [73] have investigated their integration with Transformers to enhance code understanding. Experimental results in Section IV-B show that AST-based Transformers surpass seq-based Transformers under various code transformations. However, they still struggle to capture the semantic information of transformed code, particularly under insertion / deletion, and identifier transformations, suggesting that fully leveraging ASTs for robust code semantics remains an open challenge.

The AST-based Transformer converts ASTs into node-value sequences via depth-first traversal, risking the loss of hierarchical and structural information [8], [74]. To address this, we integrate [35] into a tree-based Transformer that directly operates on the AST structure without linearization, preserving more structural information. Experimental results in Section IV-D1 demonstrate that the tree-based Transformer is more robust than the AST-based Transformer under code transformations. However, it still lacks robustness under identifier transformations, struggling to capture the semantic information of transformed code. This highlights the need for further research to better leverage AST structures for enhancing Transformers' robustness and semantic understanding.

Challenge 1: Fully leveraging Abstract Syntax Trees (ASTs) to enhance Transformers' ability to capture robust and accurate semantic information is difficult.

Considering that the robustness of tree-based Transformers arises from their ability to represent the tree structure of ASTs, future research should aim to enhance their capacity to effectively capture semantic information from ASTs. Building on this, future research could explore integrating multi-granular embeddings into Transformer models to address the challenges of leveraging ASTs for robust semantic understanding under code transformations. Specifically, integrating hierarchical embeddings from ASTs with token-level embeddings can more effectively capture both structural and contextual semantics. For example, hierarchical embeddings could be derived through tree convolution networks (TCNs) or graph neural networks (GNNs) to preserve structural relationships, combine tree representation carrying the syntactic and structural information, and source code embedding representing the lexical information to obtain the code representation.

Opportunity 1: Future work could more effectively explore techniques such as integrating multi-granular embeddings to enhance the semantic understanding of ASTs.

B. Why Does Relative Positional Encoding Affect the Robustness of Different Transformer Models Differently?

The Transformer architecture relies on self-attention, which lacks awareness of input order or structure, making positional encoding essential [34]. Prior works have demonstrated the effectiveness of various positional encoding strategies in Transformer-based code models [8], [14]. In our experiments with seq-based, AST-based, and Tree-based Transformers, using absolute and relative positional encoding, we found that relative positional encoding enhances the robustness of seq-based Transformers by improving their handling of transformed code sequences. However, it offers limited benefits for AST-based Transformers. Conversely, Tree-based Transformers show greater robustness with absolute positional encoding, likely due to the structured nature of ASTs being less compatible with relative positional encoding.

Seq-based Transformers are designed for sequential data and therefore heavily rely on token order. Relative positional encodings, which provide information about token order and distance, can improve the model's robustness. However, relative positional encoding may be less effective for structured data such as ASTs. For AST-based Transformers, the traversal order (e.g., depth-first search) provides positional information about the input, potentially reducing the importance of relative positional encoding. or tree-based Transformers, the model leverages the tree structure to better understand node contexts and relationships, while absolute positional encoding clearly indicates each node's position within the tree, which is crucial for maintaining hierarchical context. Consequently, its impact on AST-based and tree-based Transformers is either negligible or negative, highlighting the need for customized approaches to positional encoding for different types of code representations.

Future research could investigate alternative positional encoding strategies or modifications to better handle the structural complexities of ASTs in Transformer models.

C. Challenge and Potential Strategies for Insertion / Deletion Transformation

Experimental results from Sections IV-A and IV-B indicate that insertion and deletion transformations significantly affect task performance, with ID-1, ID-2, and ID-4 transformations exerting the greatest impact on Transformer robustness. These transformations introduce unrelated data, including comments, code snippets, and libraries, which substantially degrade the performance of Transformer-based code intelligence models. This underscores their vulnerability, even to simple junk code. A case study in Section IV-C5 further demonstrates that under ID-1 and ID-4, Transformers generate code summaries containing terms absent from the original program, derived from the inserted irrelevant elements. These disruptions arise from the model's inability to differentiate essential code components from non-essential ones. Overall, these findings reveal a critical limitation in Transformers' ability to prioritize relevant information, as irrelevant code misleads the models and highlights their vulnerability to minor disturbances. This weakness presents significant challenges for their practical deployment in real-world scenarios, where non-essential code elements are prevalent.

Challenge 2: The Transformer model struggles to distinguish essential code elements from non-essential ones, making it vulnerable to disruption by irrelevant code.

To address these robustness issues, future research should explore more sophisticated techniques to improve the robustness of these models. One promising approach is incorporating additional external knowledge, such as API documentation, to help the model distinguish between essential and non-essential code elements. This additional context can provide a reference point for the model, reducing the likelihood of it being misled by irrelevant information. For example, if Transformer-based models can recognize noisy parts in the input based on external information, the robustness of the models would be improved under the insertion/deletion transformation.

Another potential improvement involves developing a more effective attention mechanism that can better identify and prioritize the useful parts of the input code. Such a mechanism could enhance the model's ability to filter out irrelevant information and focus on the core semantics and structure of the code. This could be achieved through techniques like dynamic attention weights that adjust based on the relevance of the input segments. For example, integrating techniques like context-aware embeddings or hierarchical attention models might further aid in recognizing the hierarchical structure of code and discerning important elements more accurately.

Opportunity 2: Transformers need a more effective attention approach or additional external knowledge to eliminate the distraction of noisy information in the input code.

D. Challenge and Potential Strategies for Identifier Transformation

Identifiers play a crucial role in aiding models to understand code [75], [76]. Experimental results from Sections IV-A and IV-B show that identifier transformation significantly impacts all types of Transformer models across various tasks. This underscores the importance of identifiers for code intelligence tasks and the challenges Transformers face in learning code semantics under such transformations. Section IV-C4 further reveals differences in the significance of identifiers, particularly between function and variable names. For instance, renaming a single identifier, such as the main function, can notably affect code summary generation. Additionally, the case study in Section IV-C5 highlights how Transformer models rely heavily on identifiers, making them vulnerable to identifier changes. This vulnerability has been leveraged in adversarial attacks on code intelligence systems [32], [33].

Challenge 3: Transformer models heavily rely on identifiers for code intelligence tasks and are vulnerable to changes in identifiers.

Future research is encouraged to explore how to effectively understand the underlying code semantics under identifiers. For example, one promising approach is to investigate techniques to embed semantic information into identifiers, enabling models to better understand the context and purpose of identifiers beyond their surface names. Besides, researchers could explore methods to incorporate contextual information surrounding identifiers, such as their usage patterns, scope, or relationships with other code elements, to enhance the model's understanding of their semantics. Another potential improvement involves integrating human feedback mechanisms into model training and evaluation processes to validate the semantic accuracy of identifier representations and refine model performance accordingly.

Opportunity 3: Future work is expected to well exploit the underlying semantics of identifiers rather than completely rely on the literal meanings of the identifiers.

E. Non-Natural of Code Transformation

Many aspects of code, such as names, formatting, and lexical order of methods, do not impact program semantics [77]. Therefore, various semantic-preserved code transformation strategies can be applied. When code is transformed, it may lose its original structure, variable names, or comments, making it appear unnatural from a programming language perspective. In our experiments, we designed and implemented 27 and 24 code transformation strategies for Java and Python, respectively. These are rule-based code transformations, and therefore, some strategies inevitably make the original code unnatural because their transformation patterns are pre-designed. For example, the junk code insertion transformation will add code such as a new variable declaration or a non-executing loop (e.g., for i in range(0): i) into the original source code. We have designed several junk code snippets in the seed pool; the only changes are the insertion location, number of insertions, and seed selection. Thus, the naturalness of the code is inevitably lost due to rule-based code transformations.

Challenge 4: Rule-based code transformation strategy can preserve the original semantic, but it will inevitably lose the naturalness of code.

To better understand the relationship between code naturalness and model performance, more studies should investigate the specific aspects of code that contribute to its naturalness. This can include analyzing the importance of variable names, comments, and code structure. Additionally, by systematically exploring how different transformations affect model behavior, researchers can develop strategies to preserve code naturalness during preprocessing, thereby improving the robustness of Transformer models in code-related tasks. Furthermore, future work can explore advanced techniques to maintain code naturalness during transformation. For example, employing machine learning methods to predict and retain critical elements of code that contribute to its naturalness could be beneficial. These efforts will be crucial in enhancing the effectiveness of code models and ensuring their performance remains consistent even when faced with transformed code.

Opportunity 4.1: Future research should focus on identifying and preserving key elements of code naturalness, utilizing advanced techniques to maintain these characteristics during transformations.

In addition, the non-naturalness of transformed code can be a double-edged sword. It can negatively impact the model's performance, but it can also enhance performance from another perspective. For example, Chakraborty et al. [78] utilized denaturalized code to fine-tune models to understand the 'unnatural' code, helping models generate more natural code. Future work should explore how the strategic use of unnatural code can improve the effectiveness and robustness of the model. By incorporating a mix of natural and unnatural code during training, models can better handle a variety of code patterns, including those that deviate from conventional programming styles. This dual approach can enhance the model's ability to generalize and perform robustly across various coding scenarios.

Opportunity 4.2: Future work should investigate potential applications of transformed code's non-naturalness to optimize model performance and robustness by strategically balancing natural and unnatural code patterns.

Building on this perspective, the naturalness of transformed code can influence performance, as the Transformer models in our experiments are trained on a single corpus (CodeSearchNet), estimating language naturalness relative to a specific programming style or project type. For example, the experimental results in Section IV-C2 demonstrate opposite effects on Transformer models when adding or removing curly brackets from a single-statement block, suggesting that Transformer models learn specific patterns from datasets. Therefore, the performance changes in Transformer models after code transformation may be due to shifts in code naturalness, leading to patterns that differ from those in the original training dataset. However, how to qualitatively evaluate the impact of changes in code naturalness remains a challenge. Additionally, it is crucial to integrate human factors into the evaluation process [79], but conducting user studies and collecting qualitative feedback from developers can be both resource-intensive and time-consuming.

Challenge 5: Developing effective methods to evaluate how changes in code naturalness affect Transformer model performance, while considering human factors and programming style variations, remains a challenge.

To address these challenges, future research could integrate qualitative evaluation metrics with traditional performance metrics to assess the impact of changes in code naturalness on model performance and study how different coding styles affect model performance. For example, combining quantitative measures like accuracy and F1 scores with qualitative assessments of code readability and maintainability can provide researchers with a comprehensive understanding of how code naturalness affects model performance. Furthermore, collaborating with the developer community to gather feedback and real-world usage data can provide valuable insights into the practical implications of code transformations, guiding future model improvements and evaluation methodologies.

Opportunity 5: Integrating qualitative metrics with traditional performance metrics and collaborating with the developer community can provide deeper insights into the effects of code transformations.

VI. Threats to Validity

In this section, we describe the possible threats we may face in this study and discuss how we mitigate them.

Internal validity is mainly about the data prepossessing and training models. To reduce the threats, we conduct experiments based on the released code scripts, and the default training hyper-parameters for all models. Besides, we run each experiment for three times and compute the average results for all tasks.

Construct validity is mainly about the suitability of our evaluation metrics. To reduce this risk, we select the most widely used metrics for different tasks to evaluate the impact. For example, we use BLEU [64], ROUGE-L [65], and Meteor [66] to evaluate the impact of different transformations on the code summarization task.

External validity is mainly concerned with dataset we use. In our experiments, we select Java and Python datasets from CodeSearchNet. However, CodeSearchNet has some problems that will affect our experiments. For example, some instances of code in CodeSearchNet cannot be parsed into an abstract syntax tree. To reduce the threats, we filter the datasets following the previous work [70]. To further reduce the threats, we plan to collect more open-source projects to reproduce our experiments.

VII. Related Work

In this section, we introduce previous research related to our work and discuss the relevance and differences.

Divergence from adversarial robustness. Our research aims to empirically study the effect of semantic-preserving code transformations on Transformer performance from various perspectives, relating to the field of robustness in code intelligence models. However, our research differs from studies on robustness from the perspective of adversarial attacks or adversarial learning. For example, Yefet et al. [28] proposed a gradient-based optimization approach for generating adversarial attacks on code, which was effective in creating both targeted and non-targeted attacks. Zhang et al. [32] defined robust and non-robust features of DNNs and proposed an identifier renaming algorithm (Metropolis-Hastings Modifier) for generating adversarial examples on source code. Yang et al. [33] designed a black-box attack approach that considered natural semantics when generating adversarial examples of code, outperforming the method of Zhang et al. [32]. These works aim to cause the model to make different predictions via adversarial attacks, which differs from our research goal. In terms of adversarial learning, Ramakrishnan et al. [31] proposed an adversarial training method for neural models of code to enhance robustness. Bielik et al. [30] refined the representation of source code and applied adversarial training to improve the robustness of neural models while preserving high accuracy. Our research results and findings can provide deeper insights into how different code transformation strategies impact the performance of code intelligence models. Additionally, we discuss the challenges and opportunities for enhancing model robustness, aiming to create future robust code intelligence beyond just adversarial training, considering the trade-off between a model's standard accuracy and its robustness to adversarial attacks [80].

Application of code transformation. To meet our research aims, we apply semantic-preserving code transformations to transform code for the Transformer model, study performance changes, and explore how different code transformation strategies impact the performance of models. Our design for semantic-preserving code transformations is related to previous works in code transformation. Code transformations are widely used for compiler optimizations [81], [82], testability transformation [83], [84], [85], refactoring [86], [87], etc. Some works in the field of code intelligence employ code transformation, but these works differ from our research. One application of code transformation in code intelligence models is data augmentation for training or fine-tuning models to achieve more effective or robust results. For example, Yu et al. [88] applied a source-to-source transformation strategy to increase the generalization capacity of deep learning models based on program-level data augmentation. Wang et al. [89] incorporated curriculum learning into program-level data augmentation to optimize the efficiency of fine-tuning pre-trained models. Bui et al. [90] proposed a self-supervised contrastive learning framework to generate transformed code snippets and identify semantically equivalent code snippets from large-scale unlabeled data, creating an additional dataset for fine-tuning pre-trained models. Another application is adversarial attacks, but this research focuses on attack algorithms and selecting semantics-preserving samples. For example, Quiring et al. [91] used semantics-preserving code transformations to generate adversarial examples and presented a black-box attack that finds examples using Monte Carlo tree search. Li et al. [92] leveraged code transformations to attack DL-based detectors and decoupled feature learning and classifier learning to present a static analysis-based vulnerability detector. Zhang et al. [93] applied reinforcement learning to select semantics-preserving samples and proposed a black-box attack approach against GNN malware detection models.

Transformer and ASTs. Our work focuses on the robustness of Transformers in code-related tasks, particularly the utilization of Transformers on ASTs, which is similar to the study by Chirkova et al. [8], an empirical study on Transformers using ASTs. However, our research differs from this work in many aspects. First, the research aims are different. While both studies investigate Transformers using ASTs, we focus on the robustness of Transformers, whereas they aim to find effective approaches for utilizing AST structure in Transformers. Second, our study approaches are different. We utilize over 20 kinds of semantic-preserving code transformations and apply them to Transformer models to evaluate the impact; they use various AST traversal approaches for code representation to study how Transformers utilize syntactic information in ASTs. Finally, our research focus is different. We pay more attention to the impact of different code transformation strategies from various perspectives, such as different models and positional encodings; they focus on the components of ASTs (structure, node types, and values) in Transformer use. In addition, there are differences at the experimental level, including code-related task selection, dataset choice, and programming languages used, etc.

Robustness of code intelligence model. Our work is closely related to [29], which studies the generalizability of neural program models via semantic-preserving transformations. However, while our research aims are similar, our studies differ in many aspects. Our definitions of the problem are different. They define generalizability as the capability of a neural program model to return the same results under semantic-preserving transformation [29], while we define robustness as the ability of code intelligence models to maintain performance when exposed to semantic-preserving transformed code, considering modest changes in performance. This difference arises from the task selection. They focus on one task: the method name prediction task, which aims to predict the name of a method given its body. Therefore, their metrics are designed to assess whether the model predicts the same method name before and after code transformation, resulting in a binary evaluation indicator of True or False. However, our research investigates three distinct tasks: code completion, code summarization, and code search. The outputs of our code completion and code search models are lists of candidates, while the output of the code summarization model is a string with multiple tokens. Task selection influences the definition of the research problem and the evaluation metric, and it can also affect the modest changes in program behavior. For example, the degree of performance change we consider is relatively smaller than in their work. Additionally, [29] includes a limited set of six semantic-preserving transformations, while our study designs and implements 27 and 24 semantic-preserving code transformation strategies for Java and Python, respectively, grouping these strategies into five categories based on their impact scope. Finally, we focus solely on the Transformer model and study it from more various perspectives than [29], such as code transformation scope, task, programming language, positional encodings, and data structure (code vs. different AST traversals), providing a more comprehensive and extensive view of Transformer-based models.

VIII. Conclusion and Future Work

In this study, we have empirically investigated the robustness and limitations of Transformer on code intelligence. We implement 27 and 24 code transformation strategies for Java and Python languages respectively and apply the transformed code to three code intelligence tasks to study the effect on Transformer. Experimental results demonstrate that insertion / deletion transformation and identifier transformation have a great impact on Transformer's performance. Transformer based on ASTs shows more robust performance than the model based on only code sequence under most code transformations. Besides, the robustness of Transformer under code transformation is impacted by the design of positional encoding. Based on the findings, we summarize some future directions for improving the robustness of Transformer on code intelligence.

In the future, we plan to investigate more effective approaches to learn the underlying semantic information of identifiers and external knowledge to eliminate noisy information in code. Moreover, we plan to collect more open-source projects to reproduce our experiments and to support more programming languages in our code transformation.

References

[1] M. Research, "Code intelligence," Accessed: Nov. 19, 2024. [Online]. Available: https://www.microsoft.com/en-us/research/project/code-intelligence/

[2] Y. Wan et al., "Deep learning for code intelligence: Survey, benchmark and toolkit," *ACM Comput. Surveys*, vol. 56, no. 12, pp. 309:1–309:41, 2024.

[3] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2016, pp. 87–98.

[4] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. IEEE/ACM 26th Int. Conf. Program Comprehension (ICPC),* Piscataway, NJ, USA: IEEE Press, 2018, pp. 200–210.

[5] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2018, pp. 933–944.

[6] Z. Feng et al., et al., "CodeBERT: A pre-trained model for programming and natural languages," 2020, *arXiv:2002.08155*.

[7] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[8] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 703–715.

[9] T. Wolf et al., "Transformers: State-of-the-Art natural language processing," in *Proc. Conf. Empirical Methods in Natural Lang. Process.: Syst. Demonstrations*, 2020, pp. 38–45.

[10] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1433–1443.

[11] Z. Li et al., "SeTransformer: A transformer-based code semantic parser for code comment generation," *IEEE Trans. Rel.*, vol. 72, no. 1, pp. 258–273, Mar. 2023.

[12] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 150–162.

[13] M. Ciniselli et al., "An empirical study on the usage of transformer models for code completion," *IEEE Trans. Softw. Eng.*, vol. 48, no. 12, pp. 4818–4837, Dec. 2022.

[14] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2020, pp. 4998–5007.

[15] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, "Code structure guided transformer for source code summarization," 2021, *arXiv:2104.09340*.

[16] B. Berabi, J. He, V. Raychev, and M. Vechev, "TFix: Learning to fix coding errors with a text-to-text transformer," in *Proc. Int. Conf. Mach. Learn.,* PMLR, 2021, pp. 780–791.

[17] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2020, pp. 1147–1158.

[18] X. Xie et al., "DeepHunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 146–157.

[19] C.-Y. Ko, Z. Lyu, L. Weng, L. Daniel, N. Wong, and D. Lin, "POPQORN: Quantifying robustness of recurrent neural networks," in *Proc. Int. Conf. Mach. Learn.,* PMLR, 2019, pp. 3468–3477.

[20] S. Garg and G. Ramakrishnan, "BAE: BERT-based adversarial examples for text classification," 2020, *arXiv:2004.01970*.

[21] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. IEEE Symp. Secur. Privacy (SP),* Piscataway, NJ, USA: IEEE Press, 2017, pp. 39–57.

[22] D. Jakubovitz and R. Giryes, "Improving DNN robustness to adversarial attacks using jacobian regularization," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 514–529.

[23] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 303–314.

[24] T. S. Borkar and L. J. Karam, "DeepCorrect: Correcting dnn models against image distortions," *IEEE Trans. Image Process.*, vol. 28, no. 12, pp. 6022–6034, Dec. 2019.

[25] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "HotFlip: White-box adversarial examples for text classification," 2017, *arXiv:1712.06751*.

[26] X. Zheng, J. Zeng, Y. Zhou, C.-J. Hsieh, M. Cheng, and X.-J. Huang, "Evaluating and enhancing the robustness of neural network-based dependency parsing models with adversarial examples," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 6600–6610.

[27] S. Ren, Y. Deng, K. He, and W. Che, "Generating natural language adversarial examples through probability weighted word saliency," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 1085–1097.

[28] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[29] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Inf. Softw. Technol.*, vol. 135, 2021, Art. no. 106552.

[30] P. Bielik and M. Vechev, "Adversarial robustness for code," in *Proc. Int. Conf. Mach. Learn.,* PMLR, 2020, pp. 896–907.

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

[32] 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 *Proc. AAAI Conf. Artif. Intell.*, vol. 34, no. 01, 2020, pp. 1169–1176.

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

[34] G. Ke, D. He, and T.-Y. Liu, "Rethinking positional encoding in language pre-training," in *Proc. Int. Conf. Learn. Representations*, 2020.

[35] V. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 12058–12068.

[36] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," 2018, *arXiv:1803.02155*.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[38] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*.

[39] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 329–340.

[40] M. Izadi, R. Gismondi, and G. Gousios, "CodeFill: Multi-token code completion by jointly learning from structure and naming sequences," 2022, *arXiv:2202.06689*.

[41] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: AI-assisted code completion system," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery & Data Mining*, 2019, pp. 2727–2735.

[42] Y. Wang and H. Li, "Code completion by modeling flattened abstract syntax trees as graphs," in *Proc. AAAIConf. Artif. Intell.*, 2021, pp. 14015–14023.

[43] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *Proc. Int. Conf. Mach. Learn.,* PMLR, 2020, pp. 245–256.

[44] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 473–485.

[45] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2020, pp. 1385–1397.

[46] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proc. 28th Int. Conf. Program Comprehension*, 2020, pp. 184–195.

[47] Y. Zhu and M. Pan, "Automatic code summarization: A systematic literature review," 2019, *arXiv:1909.04352*.

[48] Y. Wan et al., "Improving automatic source code summarization via deep reinforcement learning," in *Proc. 33rd ACM/IEEE Int. Conf. Automat. Softw. Eng.*, 2018, pp. 397–407.

[49] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics* (Volume 1 Long Papers), 2016, pp. 2073–2083.

[50] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. Int. Conf. Learn. Representations*, 2019.

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

[52] R. Sindhgatta, "Using an information retrieval system to retrieve source code samples," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 905–908.

[53] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes, "How well do search engines support code retrieval on the web?" *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 21, no. 1, pp. 1–25, 2011.

[54] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proc. Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 964–974.

[55] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: A neural code search," in *Proc. 2nd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang.*, 2018, pp. 31–41.

[56] S. Fang, Y.-S. Tan, T. Zhang, and Y. Liu, "Self-attention networks for code search," *Inf. Softw. Technol.*, vol. 134, 2021, Art. no. 106542.

[57] J. Gu, Z. Chen, and M. Monperrus, "Multimodal representation for neural code search," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 483–494.

[58] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2019, pp. 1027–1040.

[59] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 336–347.

[60] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability transformed: Generating more accurate links with pre-trained BERT models," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 324–335.

[61] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 4159–25.

[62] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Adv. Neural Inf. Process. Syst.,* vol. 33, no. 11, pp. 1–11, 2020.

[63] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, "Code to comment "Transl." data, metrics, baselining & Eval.," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 746–757.

[64] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.

[65] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text summarization Branches Out*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2004, pp. 74–81.

[66] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. ACL Workshop Intrinsic Extrinsic Eval. Meas. Mach. Transl. Summarization*, 2005, pp. 65–72.

[67] D. R. Radev, H. Qi, H. Wu, and W. Fan, "Evaluating web-based question answering systems." in *Proc. LREC,* Citeseer, 2002.

[68] N. F. Liu et al., "Lost in the middle: How language models use long contexts," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 157–173, 2024.

[69] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 1–11.

[70] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.

[71] B. Roziere, J. M. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," 2021, *arXiv:2110.06773*.

[72] M. Szafraniec, B. Roziere, H. Leather, F. Charton, P. Labatut, and G. Synnaeve, "Code translation with compiler representations," 2022, *arXiv:2207.03578*.

[73] Z. Yang et al., "A multi-modal transformer-based code summarization approach for smart contracts," in *Proc. IEEE/ACM 29th Int. Conf. Program Comprehension (ICPC),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 1–12.

[74] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "TreeGen: A tree-based transformer architecture for code generation," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, no. 5, 2020, pp. 8984–8991.

[75] V. Efstathiou and D. Spinellis, "Semantic source code models using identifier embeddings," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR),* Piscataway, NJ, USA: IEEE Press, 2019, pp. 29–33.

[76] J. Lacomis et al., "DIRE: A neural approach to decompiled identifier naming," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2019, pp. 628–639.

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

[78] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "NatGen: Generative pre-training by "naturalizing" source code," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 18–30.

[79] T. Le-Cong, D. Nguyen, B. Le, and T. Murray, "Evaluating program repair with semantic-preserving transformations: A naturalness assessment," 2024, *arXiv:2402.11892*.

[80] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "Robustness may be at ODDS with accuracy," 2018, *arXiv:1805.12152*.

[81] K. D. Cooper et al., "Exploring the structure of the space of compilation sequences using randomized search algorithms," *J. Supercomput.*, vol. 36, no. 2, pp. 135–151, 2006.

[82] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Proc. Int. Symp. Code Gener. Optim. (CGO),* Piscataway, NJ, USA: IEEE Press, 2007, pp. 185–197.

[83] D. W. Binkley, M. Harman, and K. Lakhotia, "FlagRemover: A testability transformation for transforming loop-assigned flags," *ACM Trans. Softw. Engi. Methodol.*, vol. 20, no. 3, pp. 1–33, 2011.

[84] P. McMinn, D. Binkley, and M. Harman, "Empirical evaluation of a nesting testability transformation for evolutionary testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 3, pp. 1–27, 2009.

[85] A. Baresel, D. Binkley, M. Harman, and B. Korel, "Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach," *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 108–118, 2004.

[86] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 185–194.

[87] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[88] S. Yu, T. Wang, and J. Wang, "Data augmentation by program transformation," *J. Syst. Softw.*, vol. 190, 2022, Art. no. 111304.

[89] D. Wang et al., "Bridging pre-trained models and downstream tasks for source code understanding," 2021, *arXiv:2112.02268*.

[90] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proc. 44th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2021, pp. 511–521.

[91] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *Proc. 28th* {USENIX} Secur. Symp. ({USENIX} Secur.), 2019, pp. 479–496.

[92] Z. Li et al., "Towards making deep learning-based vulnerability detectors robust," 2021, *arXiv:2108.00669*.

[93] L. Zhang, P. Liu, and Y.-H. Choi, "Semantic-preserving reinforcement learning attack against graph neural networks for malware detection," 2020, *arXiv:2009.05602*.