

Two Sides of the Same Coin: Exploiting the Impact of Identifiers in Neural Code Comprehension

Shuzheng Gao¹, Cuiyun Gao^{1*}, Chaozheng Wang¹, Jun Sun², David Lo², Yue Yu³

¹ School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China

² Singapore Management University, Singapore

³ National University of Defense Technology

szgao98@gmail.com, gaocuiyun@hit.edu.cn, wangchaozheng@stu.hit.edu.cn, junsun@smu.edu.sg, davidlo@smu.edu.sg, yuyue@nudt.edu.cn

Abstract—Previous studies have demonstrated that neural code comprehension models are vulnerable to identifier naming. By renaming as few as one identifier in the source code, the models would output completely irrelevant results, indicating that identifiers can be misleading for model prediction. However, identifiers are not completely detrimental to code comprehension, since the semantics of identifier names can be related to the program semantics. Well exploiting the two opposite impacts of identifiers is essential for enhancing the robustness and accuracy of neural code comprehension, and still remains under-explored. In this work, we propose to model the impact of identifiers from a novel causal perspective, and propose a counterfactual reasoning-based framework named CREAM. CREAM explicitly captures the misleading information of identifiers through multi-task learning in the training stage, and reduces the misleading impact by counterfactual inference in the inference stage. We evaluate CREAM on three popular neural code comprehension tasks, including function naming, defect detection and code classification. Experiment results show that CREAM not only significantly outperforms baselines in terms of robustness (e.g., +37.9% on the function naming task at F1 score), but also achieve improved results on the original datasets (e.g., +0.5% on the function naming task at F1 score).

I. INTRODUCTION

With the rapid development of artificial intelligence techniques, automated code comprehension has drawn more and more attention in the software engineering community [1]–[5]. Recently, many neural code comprehension models have been proposed to learn the code semantics with deep neural networks (DNNs), and achieved state-of-the-art performance on various tasks, such as code summarization [3], [6], [7], function naming [8]–[10], and defect detection [11]–[14].

Despite these success cases, recent studies [15], [16] show that the neural code comprehension models are sensitive to identifier naming. On the one hand, identifier names can be misleading for model prediction [15], [17]–[19]. For example, as illustrated in Figure 1 (a), given the original code snippet, the popular function naming model NCS [20] correctly predicts the function name as “sort”. However, if the identifier name “arr” is renamed to “f”, the model outputs an irrelevant

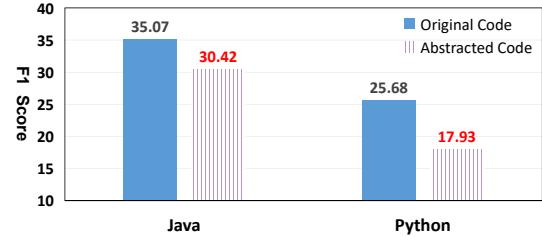
* Corresponding author. The author is also affiliated with Peng Cheng Laboratory and Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies.

```

1 public static void □ (int[] arr) {
2     for (int i = 1; i < arr.length; i++){
3         int x = arr[i];
4         int j = i - 1;
5         while (j >= 0 && arr[j] > x) {
6             arr[j + 1] = arr[j];
7             j--;
8         }
9     }
10    arr[j + 1] = x;   Ground Truth: sort
11 }   Prediction: open

```

(a) An example for illustrating that identifiers can be misleading for neural code comprehension models. By changing the identifier name “arr” to “f”, the model outputs a wrong result.



(b) Results illustrating that code abstraction leads to performance drop on F1 score of the task.

Fig. 1: Illustration of the opposite impact of identifiers on the function naming task.

function name “open”. On the other hand, identifier names are not completely harmful to code comprehension and can provide rich information about the code semantics [21]. As shown in [17], [21], discarding identifier semantics results in poor model performance. As illustrated in Figure 1 (b), the code abstraction technique [21], [22], which renames all the identifiers with placeholders (e.g., “VAR_0”), leads to an obvious performance degradation on the prediction, showing 13.26% and 30.18% drop in terms of the F1 score for Java and Python, respectively. Therefore, the robustness issue caused by identifier names and their useful semantics are two sides of the same coin. Properly leveraging the useful information and alleviating the misleading impact of identifiers are essential for building an accurate and robust code comprehension model, which is a problem that remains under-explored.

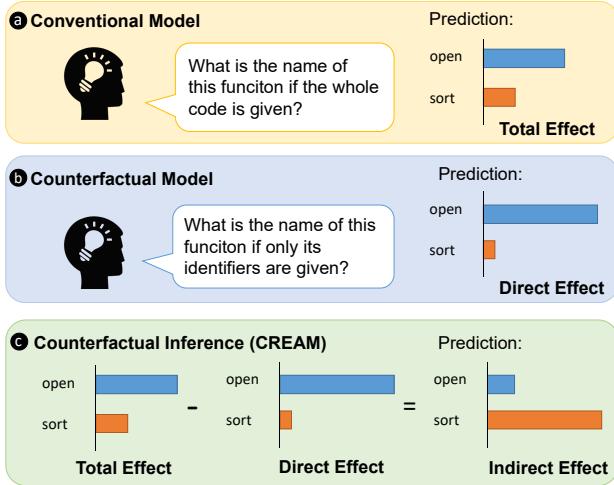


Fig. 2: Our causal view on the impact of identifiers in neural code comprehension. The conventional model (a) and counterfactual model (b) estimate the result of corresponding question, respectively. Counterfactual inference (c) eliminates the *direct effect* of identifiers, and gets more accurate and robust prediction. The prediction results are from the example in Figure 1 (a).

However, it is challenging to well balance the useful and misleading impact of identifiers on the model performance. The main challenge is that it is difficult to explicitly distinguish the two opposite impacts, the deep learning models are black-box at this stage [23]. Besides, although producing more training instances might be helpful, the process would require non-trivial manual labeling efforts and extra training costs [24], [25]. In this work, we aim at exploiting the impact of identifiers without the cost of more training data.

Inspired by causal inference from the statistics field [26], which has proven effective in analyzing opposite impacts in fields such as psychology [27] and politics [28], we propose to solve the problem from a causal view. Specifically, the impact of the identifiers on model prediction can be divided into two separate causal effects, *direct effect* and *indirect effect*. By formulating the useful and misleading information as the *direct effect* and *indirect effect* respectively, we propose to preserve the useful information and mitigate the misleading impact of identifiers names by counterfactual inference [29], [30], i.e. subtracting the direct identifier effect from the total effect. As illustrated in Figure 2 (a), conventional models perform prediction by estimating the *total effect* of the input code snippet, and thereby are sensitive to identifier names. From the causal perspective, we first compute the *direct effect* of identifiers by estimating “*what the name of this function would be if only its identifiers were given?*”, as shown in Figure 2 (b). The *direct effect* is then removed through counterfactual inference for alleviating the misleading information of identifiers, as depicted in Figure 2 (c).

In this work, we propose CREAM, a Counterfactual REAsoning-based fraMework (CREAM) to exploit the impact

of identifiers in neural code comprehension. Specifically, in the training stage, CREAM distinguishes and captures *direct effect* and *indirect effect* by multi-task learning. In the inference stage, CREAM eliminates the misleading impact by reducing the direct identifier effect from the prediction. We evaluate the performance of CREAM on three popular code comprehension tasks, including function naming, software defect detection and code classification. For each task, we choose no fewer than three popular conventional models and enhance them with the proposed framework CREAM. For evaluating the robustness improvement brought by removing the misleading information of identifiers, we establish a transformed test set for each dataset by randomly substituting each identifier in the original test set with another one in the dataset. Experimental results demonstrate that the models equipped with CREAM not only significantly improve robustness on the transformed test sets but also achieve improved performance on the original test sets. Specifically, CREAM improves the robustness of CodeBERT on the transformed test set by 37.9%, 8.3% and 1.9% on function naming, defect detection, and code classification respectively; while on the original test set, CREAM achieves an improvement of 0.5%, 0.9%, and 0.3% on function naming, defect detection, and code classification respectively.

The main contributions of this paper are summarized as follows:

- 1) To the best of our knowledge, we are the first to exploit the two-sided impact of identifiers in neural code comprehension from a causal view.
- 2) We propose a novel counterfactual reasoning-based framework CREAM for capturing the misleading information of identifiers via multi-task learning, and mitigating the misleading information via counterfactual inference.
- 3) Extensive experiments show that the proposed framework CREAM is more robust to identifier renaming than conventional models on various code comprehension tasks. The removal of misleading impact also improves the overall performance of CREAM on the original datasets.

II. PRELIMINARIES

In this section, we review the key concepts we used in causal inference. In the following, we use the capital letter (e.g., T) and lowercase letter (e.g., t) to denote a variable and a specific value respectively.

Structural Causal Model. The Structural Causal Model (SCM) [26], [31] reflects the causal relations between variables through a directed acyclic graph $G = \{V, E\}$, where V denotes the set of variables and E denotes the set of edges that describe the direct causal relationship between variables. These causal relationships can be further parameterized with *structural equation* [32]. Figure 3 is a simple example of SCM involving three variables, *treatment variable* medicine (M), *mediator variable* placebo effect (P) and *outcome variable* disease (D) [26]. Their causal relationships can be presented as follows:

$$P_m = f_P(M = m), \quad (1)$$

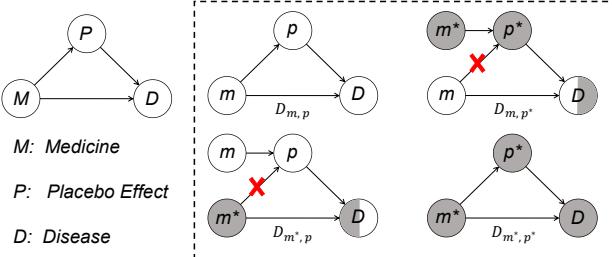


Fig. 3: An example of SCM. White and gray nodes denote the variables are at the value of factual and counterfactual status respectively. Counterfactual notations $D_{m,p}$, $D_{m^*,p}$, D_{m,p^*} and D_{m^*,p^*} are illustrated in the four graphs on the right.

$$D_{m,p} = f_D(M = m, P = p), \quad (2)$$

where f_P and f_D denote the *structural equation* of corresponding variable, respectively. In this case, the causal effect of medicine on disease exists in two paths. The first path $M \rightarrow D$ denotes that medicine has a direct effect on disease through the biological mechanism. The other path $M \rightarrow P \rightarrow D$ shows that taking medicines can also alleviate the disease through the mediator, placebo effect. Thus when estimating to what extent the medicine affects the disease through the placebo effect, we need to exclude the bias caused by the biological mechanism of medicine, i.e., $M \rightarrow D$.

Counterfactual Inference. Counterfactual inference [26], [31] is used to estimate for the same individual what the outcome variable would be if the value of some variables were different from the value we observed in the reality. As shown in Figure 3, counterfactual inference can answer the following question: *whether the disease of the patient could be alleviated if he didn't receive the placebo effect but took medicines*. Specifically, it estimates the value of D when P receives $M = m$ through $M \rightarrow P$, while D receives $M = m^*$ through $M \rightarrow D$. Here we use the asterisk notation to represent the situation where the value of the node is muted from the reality, e.g., p^* denotes that the patient did not receive placebo effect. This estimation can be achieved by using $do(P = p^*)$:

$$D_{m,p^*} = f_D(M = m, do(P = p^*)), \quad (3)$$

where $do(\cdot)$ operator denotes the intervention defined by SCM [26], [33]. It forcibly substitutes $p = f_P(M = m)$ with $p^* = f_P(M = m^*)$ in the structural equation f_D . Note that $do(P = p^*)$ does not affect the ascendant variable of P , i.e., M retains its value m on the direct path $M \rightarrow D$. In clinical trials, it represents that the patient takes medicine without being informed.

Causal effects. The causal effects measure to what extent value change of the treatment variable (e.g., the value of M change from m^* to m) affects the value of the outcome variable (e.g., D). For example in Figure 3, the *total effect* (TE) [26] of M on D is defined as:

$$TE = D_{m,p} - D_{m^*,p^*}, \quad (4)$$

where D_{m^*,p^*} denotes the situation that the patient neither took medicines nor received the placebo effect. We can see that TE calculates the causal effect of M to D from both the direct causal path $M \rightarrow D$ and the indirect causal path $M \rightarrow P \rightarrow D$. For a detailed analysis, existing work often decomposes TE into *natural direct effect* (NDE) and *total indirect effect* (TIE) through $TE = NDE + TIE$ [29], [34]. NDE represents the value change of the outcome variable when value change of the treatment variable only affects it through the direct path $M \rightarrow D$. Formally, NDE is defined as follows:

$$NDE = D_{m,p^*} - D_{m^*,p^*}, \quad (5)$$

Accordingly, the TIE can be obtained by subtracting NDE from TE:

$$TIE = TE - NDE = D_{m,p} - D_{m^*,p^*}, \quad (6)$$

TIE measures value change of the outcome variable when value change of the treatment variable only affects it through the indirect path $M \rightarrow P \rightarrow D$. Equipped with the above causality concepts, we solve the problem of estimating the influence of placebo effect via calculating TIE of M on D .

III. METHODOLOGY

In this section, we first present our causal view of the prediction process for neural code comprehension models, during which the impact of identifiers on the model prediction is formulated with an SCM. Then we describe our proposed CREAM framework for preserving the useful impact while eliminating the misleading impact of identifiers.

A. A Causal View on Neural Code Comprehension

As shown in Figure 4 (a), we abstract the prediction process of neural code comprehension by defining four variables: 1) **naming information T** , which denotes the code tokens related to identifier naming, i.e., user-defined identifiers; 2) **non-naming information F** , which denotes the code properties irrelevant to identifier naming, i.e., the code tokens other than the identifiers; 3) **combined knowledge K** , which serves as a mediator exploiting both the naming information and non-naming information for model prediction; 4) **model prediction R** , which denotes the prediction results of code comprehension tasks, e.g., classification scores for code classification tasks.

Based on the above variable definitions and SCM introduced in Section II, we formulate the causal structure of conventional models, as illustrated in Figure 4 (a). The causal relationships are shown as follows:

- 1) $F \rightarrow R$ represents the causal relationship from non-naming information F to the model prediction R . For the example shown in Figure 1 (a), the code structure which contains two loops corresponds to the variable F . Since the code structure is about data sorting-related swap operations, it is helpful for predicting the function name as “sort”. We regard this path as a “beneficial” path, since the prediction based on only the non-naming information is robust to identifier renaming.

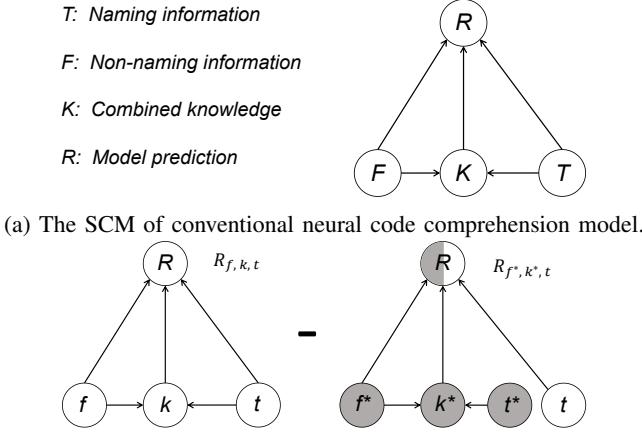


Fig. 4: Illustration for the SCM of conventional neural code comprehension model (a), and our counterfactual reasoning-based neural code comprehension model (b).

- 2) $T \rightarrow R$ represents the causal relationship from naming information T to the model prediction R . For the example in Figure 1 (a), the identifier “ f ” which corresponds to variable T , misleads the model to predict “*open*” as the function name. The possible reason of the wrong prediction is that the identifier “ f ” commonly appears in the functions named “*open*” in the training set. For example, in a Java function named “*open*”, “ f ” is usually used as an abbreviation of a file handling object, e.g., “*File f = new File()*”. The phenomenon is also called *spurious correlation* [26] between identifiers and prediction results. Since the *spurious correlation* will mislead models’ understanding of the code semantics, we regard this direct causal effect as the misleading impact of identifiers.
- 3) $F, T \rightarrow K \rightarrow R$ represents the process that models predict based on the combined knowledge K which exploits both the naming information T and non-naming information F . We also regard this path as a “beneficial” path since it not only leverages the robust non-naming information F and also the useful information in identifier names T . Although identifier names are essentially irrelevant to program behaviors, the semantics are helpful for accurate code comprehension [17], [21]. The path represents that we leverage the useful information of identifiers through the combined knowledge K instead of completely discarding the identifiers.

As shown in Figure 4 (a), conventional neural code comprehension models predict through the *total effect*, i.e., $R_{f,k,t} - R_{f^*,k^*,t^*}$, which integrates the *direct effect* from all the three paths. In this way, the inclusion of the “bad” path $T \rightarrow R$ will inevitably introduce the misleading impact of identifiers to the prediction of conventional models. Thus, to improve the robustness and accuracy of the models, the *direct effect* of $T \rightarrow R$ from the *total effect* should be excluded

during model prediction. To achieve the goal, we introduce the counterfactual reasoning-based neural code comprehension model which estimates the causal effect of T and F on the prediction R with *direct effect* $T \rightarrow R$ blocked, as shown in Figure 4 (b). We describe how we implement the idea and details of the proposed general framework CREAM for eliminating the misleading impact of identifiers in the next section.

B. Details of CREAM

In this section, we elaborate on the details of CREAM. Figure 5 (a) and (b) illustrate the overall workflow of the conventional model and the proposed CREAM, respectively. For conventional models, the training and inference stages are the same; while for CREAM, the calculation of classification scores¹ in the training and inference stage are different. Specifically, in the training stage, CREAM captures each *direct effect* in SCM through multi-task learning; while in the inference stage, it eliminates the misleading impact of identifiers by counterfactual inference.

1) *Task formulation*: In this work, we broadly divide neural code comprehension tasks into two paradigms, i.e., classification-based and generation-based. Considering that the generation-based task can be viewed as a successive classification task, where the models output classification scores over the vocabulary at each time step, we unify the workflow of the neural code comprehension tasks as follows. Assume that we have a source code database $X = \{x_1, x_2, \dots, x_n\}$ and corresponding ground truth $Y = \{y_1, y_2, \dots, y_n\}$, where n is the number of training data and y_j is a class label or target sequence of the j -th training instance for classification and generation task, respectively. In the training stage, our goal is to train a neural model \mathcal{F} to minimize the prediction error on the training set. Formally, \mathcal{F} can be formulated as:

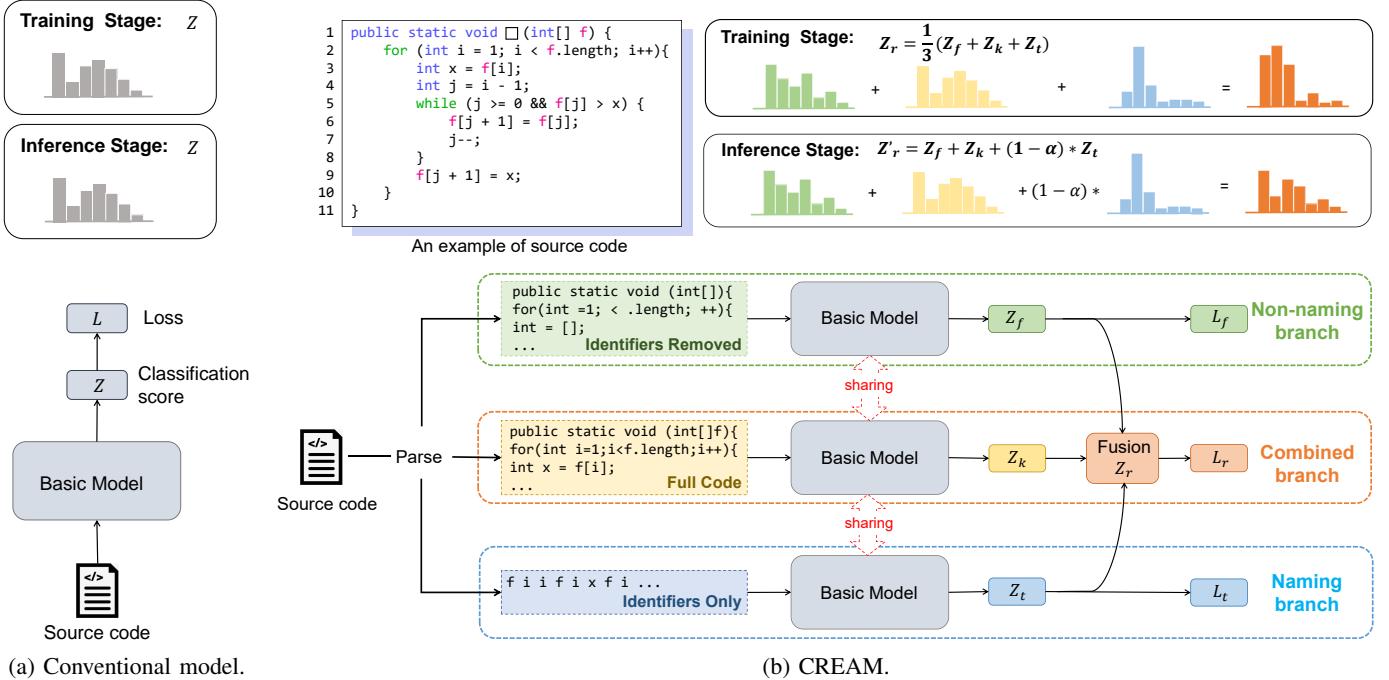
$$\bar{\mathcal{F}} = \arg \min_{\mathcal{F}} \sum_{(x,y) \in \{X,Y\}} L(\mathcal{F}(x), y), \quad (7)$$

where $L(\cdot)$ denotes the loss function such as cross entropy and \mathcal{F} can be a large variety of existing neural models such as Long Short-Term Memory (LSTM) [37] and Transformer [38]. In the inference stage, given a sample x' in the test set, the model first calculates the classification score, i.e., Z and Z'_r for the conventional model and CREAM, respectively (as shown in Figure 5). Following the widely-used greedy search strategy, the class with the highest classification score is selected as prediction result:

$$y_r = \arg \max_c (z'), \quad (8)$$

where c is the set of candidate classes and y_r is the prediction result.

¹It is also called logits in many machine learning papers [35], [36].



(a) Conventional model.

Fig. 5: The overall workflow of conventional neural code comprehension model (a) and CREAM (b). The basic model in (a) and (b) denotes the conventional code comprehension models.

2) *Framework Design*: As shown in Figure 5 (b), our proposed CREAM framework follows the SCM illustrated in Section III-A. Specifically, we distinguish the three causal paths including $F \rightarrow R$, $K \rightarrow R$ and $T \rightarrow R$ by designing three branches in CREAM, i.e., non-naming branch, combined branch, and naming branch, respectively. To avoid increasing the model size, the parameters of the basic models for the three branches are shared. CREAM first parses the input source code into three types of input, including non-naming input f , combined input k , and naming input t for the three branches, respectively. CREAM then estimates the direct causal effect of each path by calculating the classification score for each branch based on the basic model, formulated as:

$$Z_f = \mathcal{F}(f), Z_k = \mathcal{F}(k), Z_t = \mathcal{F}(t). \quad (9)$$

To obtain the *total effect* of the input code on prediction, we combine the direct causal effect from three branches and fuse their outputs into the final classification score Z_r which is corresponding to the variable R in SCM. Here we compute Z_r by an averagely weighted method

$$Z_r = \frac{1}{3}(Z_f + Z_k + Z_t), \quad (10)$$

According to the definition of *structural equation* in Section II, the fusion method also indicates that the *structural equation* of the variable R is parameterized as follows:

$$R_{f,k,t} = \frac{1}{3}(\mathcal{F}(f) + \mathcal{F}(k) + \mathcal{F}(t)). \quad (11)$$

Based on the above framework design of CREAM, we introduce the computation process which includes two stages, i.e.,

training stage and inference stage. The details are illustrated in Algorithm 1.

Multi-task Training: In the training stage, CREAM needs to realize multiple goals: 1) to accurately estimate the *total effect* of source code on prediction results; and 2) to capture the *direct effect* of each path and distinguish it from *total effect*. To achieve the goals, we adopt the multi-task learning strategy [30], [39] for model training:

$$L_f = L(Z_f, y), L_r = L(Z_r, y), L_t = L(Z_t, y), \quad (12)$$

$$L_{total} = L_f + L_r + L_t, \quad (13)$$

Here, L_r is used to train the model for accurately estimating the *total effect* (i.e., the first goal); while L_f and L_t are involved to capture the *direct effect* and distinguish it from the *total effect* (i.e., the second goal). Besides, to simultaneously capture the *direct effect* and distinguish it from *total effect*, we also adopt the deferred training strategy [40]. Specifically, we first train the three branches separately to well capture each *direct effect*, and then fuse their classification scores after I_{fusion} iterations (Lines 4-7 of the multi-task training stage in Algorithm 1).

Counterfactual Inference: As illustrated in Section III-A, the key to eliminate the misleading impact of identifiers is to remove the *direct effect* $T \rightarrow R$ from the *total effect*. To this end, during the inference stage, CREAM first estimates the *total effect* of the input code on model prediction R as follows:

$$TE = R_{f,k,t} - R_{f^*,k^*,t^*}, \quad (14)$$

where f^* , k^* and t^* denote the corresponding empty input. Following [29], [30], we set the classification score to a uniform distribution (i.e., the classification scores for all classes are the same) if the input is empty, which is formulated as:

$$\mathcal{F}(f^*) = \mathcal{F}(k^*) = \mathcal{F}(t^*) = u \quad (15)$$

where u is the classification score under uniform distribution. Then CREAM estimates the misleading impact by calculating the *natural direct effect* of identifier names on model prediction, i.e., the *direct effect* of $T = t$ on prediction R under the situation $F = f^*$ and $K = k^*$:

$$NDE = R_{f^*,k^*,t} - R_{f^*,k^*,t}. \quad (16)$$

We finally eliminate the misleading impact by subtracting the *natural direct effect* from the *total effect*. We propose to control the degree of elimination by involving a hyper-parameter α , defined as:

$$TE - \alpha * NDE \propto R_{f,k,t} - \alpha * R_{f^*,k^*,t}, \quad (17)$$

where α ranges from 0 to 1. By omitting the constant, the final classification score Z'_r is calculated as follows:

$$\begin{aligned} Z'_r &= R_{f,k,t} - \alpha * R_{f^*,k^*,t^*} \\ &= \frac{1}{3}(\mathcal{F}(f) + \mathcal{F}(k) + \mathcal{F}(t)) - \frac{\alpha}{3}(\mathcal{F}(f^*) + \mathcal{F}(k^*) + \mathcal{F}(t)) \\ &= \frac{1}{3}(\mathcal{F}(f) + \mathcal{F}(k) + \mathcal{F}(t)) - \frac{\alpha}{3}(u + u + \mathcal{F}(t)) \\ &\propto \mathcal{F}(f) + \mathcal{F}(k) + (1 - \alpha) * \mathcal{F}(t) \\ &= Z_f + Z_k + (1 - \alpha) * Z_t. \end{aligned} \quad (18)$$

IV. EXPERIMENTAL SETUP

In this section, we detail the experimental settings for the three popular code comprehension tasks including function naming, defect detection and code classification, which have been active areas of software engineering research for years.

A. Evaluation Tasks

1) *Function naming*: Function naming aims to automatically generate a meaningful and succinct name for a function. In software industry, it can help engineers correct the inconsistent method and API name for program readability and maintainability [10], [41], [42]. In this work, we formulate it as a generation task with the greedy search strategy and cross-entropy loss function.

2) *Defect detection*: Given a code snippet, defect detection aims to identify whether a given code snippet is vulnerable, which is crucial to defend a software system from cyberattack [11], [12]. In the previous work, it is formulated as a binary classification task and generally uses the binary cross-entropy [43] as the loss function.

3) *Code classification*: Code classification is the task of classifying a code snippet by its functionality, which is helpful for program comprehension and maintenance [1], [44]. It is formulated as a multi-class classification task and utilizes cross-entropy as the loss function.

Algorithm 1 Algorithm of CREAM framework

Input: training set $\{X_{train}, Y_{train}\}$, test set $\{X_{test}\}$, fusion iteration threshold I_{fusion} , total iteration I

Output: neural model \mathcal{F} , prediction result y_r

Multi-task Training:

```

1: for  $i \in \{1, \dots, I\}$  do
2:   Extract  $f, t, k$  from  $X_{train}$ 
3:    $Z_f = \mathcal{F}(f), Z_t = \mathcal{F}(t), Z_k = \mathcal{F}(k)$ 
4:   if  $i \leq I_{fusion}$  then
5:      $Z_r = Z_k$ 
6:   else
7:      $Z_r = \frac{1}{3}(Z_f + Z_k + Z_t)$ 
8:   end if
9:   Calculating  $L_r, L_t, L_f$  with  $Y_{train}$ 
10:  Update model  $\mathcal{F}$  with  $L_{total} = L_r + L_t + L_f$ 
11: end for
12: return model  $\mathcal{F}$ 

```

Counterfactual Inference:

```

1: Extract  $f, t, k$  from  $X_{test}$ 
2:  $Z_f = \mathcal{F}(f), Z_t = \mathcal{F}(t), Z_k = \mathcal{F}(k)$ 
3:  $Z_r = Z_f + Z_k + (1 - \alpha) * Z_t$ 
4:  $y_r = \arg \max(Z_r)$ 
5: return Prediction result  $y_r$ 

```

B. Baselines

1) *Function naming*: For function naming, we adopt three well-known code-to-text models for evaluation. **CodeNN** [6] is a classical sequence-to-sequence model which generates source code summaries with an LSTM network and attention mechanism. **NCS** [20] is a recent state-of-art-model on code summarization. **CodeBERT** [45] is a widely-used pre-trained model for source code. We fine-tune CodeBERT with the pre-trained encoder with an additional decoder training from scratch.

2) *Defect detection*: For defect detection, we follow the popular benchmark CodeXGLUE [46] and adopt the following models. **TextCNN** [47] and **BiLSTM Att** [37] are two widely used methods for text classification in NLP. Here, **BiLSTM Att** is the combination of BiLSTM and attention mechanism [48]. Many defect detection works [49], [50] employ them for model construction. **Devign** [11] is proposed to learn the various vulnerability characteristics with a composite code property graph and graph neural network. We also use **CodeBERT** as the basic model since it has shown promising results on defect detection [46].

3) *Code classification*: We adopt three representative works in this field as the basic model for CREAM. **TBCNN** [44] is a classical code classification model which captures structural information of the Abstract Syntax Tree (AST) with a tree-based convolution neural network. **ASTNN** [11] learns the code representation by splitting the large AST into a sequence of small statement trees. We also involve the pre-trained model **CodeBERT** for evaluation.

TABLE I: Statistics of the benchmark datasets.

Datasets	Train	Validation	Test
CSN-Java	164,923	5,183	10,955
CSN-Python	251,820	13,914	14,014
CSN-Go	167,288	7,325	8,122
CSN-PHP	241,241	12,982	14,014
CSN-Ruby	24,927	1,400	1,261
CSN-JavaScript	38,499	2,745	2,232
Defect Detection	21,854	2,732	2,732
Code Classification	31,200	10,400	10,400

C. Datasets and Metrics

1) *Function naming*: For function naming, we use the widely used CodeSearchNet (CSN) [51] dataset which contains six programming languages including Java, Python, Go, PHP, JavaScript and Ruby. Specifically, we use the cleaned dataset which is pre-processed and open sourced in CodeBERT [45]. For JavaScript, we further filter the samples without a function name. To measure the similarity between generated function names and the reference names, we employ the standard metrics including Precision, Recall and F1.

2) *Defect detection*: We use the defect detection dataset released by Devign [11]. The dataset contains 27,318 C code snippets collected from the QEMU and FFmpeg projects. As for the dataset split, we use the benchmark open sourced by CodeXGLUE [46], in which the dataset is split into training set, validation set and test set in a proportion of 8:1:1. Following [11], [52], we use accuracy as the evaluation metric.

3) *Code classification*: For code classification, we use the POJ dataset [44] which contains 52,000 code snippets of C language with 104 classes. It is collected from Online Judge (OJ) and code snippets in the same class are used to solve the same programming problem. We follow ASTNN [11] to split the dataset into training set, validation set and test set in a proportion of 3:1:1. We follow previous work [1], [44] in this field and use accuracy as the evaluation metric.

We list the statistics of the benchmark datasets in Table I. When evaluating the robustness, we follow previous work [53]–[55] and validate whether the model can output the same results under semantic-preserving code transformations, i.e., identifier renaming. Therefore we also create a transformed test set for each dataset. Specifically, following the procedure in previous work [53]–[55], we randomly substitute the identifier names in the test set with another identifier name that appeared in the dataset.

D. Implementation Details

During the experiment, we reproduce each model either directly using the code released by the author or strictly following the steps described in their paper². For a fair comparison, we make sure that the hyperparameters such as training epochs and learning rate for models with and without CREAM are exactly the same. The value of α in Equ. (17) is set as 0.4, 0.5, 0.6, 0.7 or 0.8 for different datasets. The I_{fusion}

²For the baseline Devign in the defect detection, we reproduce the method based on the re-implementation code in [56] due to the lack of original code.

is set as 10% of total training iterations. We will discuss how we select parameters for each dataset in Section V-D.

When applying CREAM to each baseline, we parse the source code into three sequences (Figure 5) for the models that treat source code as plain text (e.g., NCS and CodeBERT) for the function naming task. For models that treat the source code as a tree [1] or graph [11], we divide the tree or graph into two parts which contain the nodes with and without the naming information, respectively. To extract the identifiers in source code, we first parse the code into AST with tree-sitter³, and filter the leaf node according to its type and the type of its parent. For example, we extract the identifiers for C/C++ by selecting the leaf node whose type is “*identifier*” and parent’s type is not “*call_expression*”.

All the experiments are conducted on a server with 4 Nvidia Tesla V100 GPUs and 32 GB graphic memory. We run each baseline and CREAM three times and report the best results.

V. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of CREAM by answering the following research questions:

RQ1: Does CREAM improve the robustness of existing models?

RQ2: Is CREAM beneficial for improving the accuracy of existing models?

RQ3: What is the impact of multi-task learning and counterfactual inference on the performance of CREAM?

RQ4: How do different parameter settings affect the performance of CREAM?

A. RQ1: Evaluation on the Robustness of CREAM

We evaluate the robustness of CREAM on the transformed datasets for the three tasks, with results illustrated in Table II–IV, respectively. Due to the page limit, we only present the F1 scores of different models for the function naming task in Table II. The scores for the precision and recall metrics are presented on the GitHub repository⁴. Based on the results, we have the following observations:

Conventional models are not robust to identifier naming. By comparing the results of conventional models on the original test set and transformed test set, we find that identifier renaming leads to significant performance degradation on the three tasks. For example, the average performance of the models for function naming, defect detection, and code classification drop by 55.4%, 8.9% and 12.3%, respectively. The results indicate that conventional neural code comprehension models are easily misled by identifier names.

CREAM consistently and significantly improves the model robustness. As can be seen in Table II–IV, CREAM consistently outperforms the conventional models on all the task datasets, demonstrating its capability and generalizability in improving the robustness of neural code comprehension models. Besides, the improvement over the conventional models is substantial, for example, for function naming CREAM

³<https://github.com/tree-sitter/tree-sitter>

⁴<https://github.com/ReliableCoding/CREAM>

TABLE II: Experimental results (F1 score) on function naming. Percentages listed within parentheses are computed improvement or reduction in F1 score as compared with the results of the corresponding basic model. “*” denotes statistical significance in comparison to the baselines (i.e., two-sided t -test with p -value < 0.05)

Approach	Java	Python	JavaScript	PHP	Go	Ruby	Average
Transformed test set							
CodeNN	23.05	4.00	4.72	17.61	17.79	5.98	12.19
+CREAM	28.35 ($\uparrow 22.99\%$)*	5.44 ($\uparrow 36.00\%$)*	6.00 ($\uparrow 27.12\%$)*	22.44 ($\uparrow 27.43\%$)*	24.91 ($\uparrow 40.02\%$)*	7.25 ($\uparrow 21.24\%$)*	15.73 ($\uparrow 29.04\%$)*
NCS	21.61	3.65	3.32	17.56	21.68	5.70	12.25
+CREAM	27.28 ($\uparrow 26.24\%$)*	4.67 ($\uparrow 27.95\%$)*	7.20 ($\uparrow 116.87\%$)*	24.38 ($\uparrow 38.84\%$)*	27.33 ($\uparrow 26.06\%$)*	10.47 ($\uparrow 83.68\%$)*	16.89 ($\uparrow 37.88\%$)*
CodeBERT	25.90	5.05	4.61	23.43	27.72	20.72	17.91
+CREAM	36.72 ($\uparrow 41.78\%$)*	7.29 ($\uparrow 44.36\%$)*	10.54 ($\uparrow 128.63\%$)*	35.80 ($\uparrow 52.80\%$)*	38.62 ($\uparrow 39.32\%$)*	27.49 ($\uparrow 32.67\%$)*	26.08 ($\uparrow 45.62\%$)*
Original test set							
CodeNN	33.18	22.64	14.40	36.83	34.64	12.24	25.66
+CREAM	33.69 ($\uparrow 1.54\%$)*	22.69 ($\uparrow 0.22\%$)*	14.16($\downarrow 1.67\%$)	36.87 ($\uparrow 0.11\%$)	34.79 ($\uparrow 0.43\%$)*	12.36 ($\uparrow 0.98\%$)*	25.76 ($\uparrow 0.39\%$)*
NCS	35.07	25.68	16.31	40.19	40.52	12.93	28.45
+CREAM	35.63 ($\uparrow 1.59\%$)*	25.40($\downarrow 1.09\%$)	16.46 ($\uparrow 0.92\%$)	40.25 ($\uparrow 0.04\%$)	40.21($\downarrow 0.77\%$)	13.64 ($\uparrow 5.5\%$)*	28.60 ($\uparrow 0.53\%$)*
CodeBERT	46.38	37.64	29.55	49.36	49.88	32.04	40.81
+CREAM	47.04 ($\uparrow 1.42\%$)*	37.66 ($\uparrow 0.05\%$)	27.11($\downarrow 8.26\%$)	50.35 ($\uparrow 2.01\%$)*	50.28 ($\uparrow 0.80\%$)*	30.49($\downarrow 4.84\%$)	40.49($\downarrow 0.78\%$)

TABLE III: Experimental results (accuracy) on defect detection. “*” denotes statistical significance in comparison to the baselines (i.e., two-sided t -test with p -value < 0.05)

Approach	Original test set	Transformed test set
TextCNN	58.57	54.36
+CREAM	59.96 ($\uparrow 2.37\%$)*	55.78 ($\uparrow 2.61\%$)*
BiLSTM Att	62.08	54.90
+CREAM	62.34 ($\uparrow 0.42\%$)*	56.55 ($\uparrow 3.01\%$)*
Devign	55.77	52.03
+CREAM	56.74 ($\uparrow 1.74\%$)*	53.98 ($\uparrow 3.75\%$)*
CodeBERT	63.47	57.24
+CREAM	64.05 ($\uparrow 0.91\%$)*	62.01 ($\uparrow 8.33\%$)*

TABLE IV: Experimental results (accuracy) on code classification. “*” denotes statistical significance in comparison to the baselines (i.e., two-sided t -test with p -value < 0.05)

Approach	Original test set	Transformed test set
TBCNN	96.76	68.45
+CREAM	97.00 ($\uparrow 0.24\%$)*	83.08 ($\uparrow 21.37\%$)*
ASTNN	98.04	89.43
+CREAM	98.18 ($\uparrow 0.14\%$)*	96.06 ($\uparrow 7.41\%$)*
CodeBERT	97.96	95.76
+CREAM	98.28 ($\uparrow 0.33\%$)*	97.56 ($\uparrow 1.88\%$)*

improves the performance of CodeNN, NCS and CodeBERT on the transformed test set by 29.0%, 37.9%, and 45.6%, respectively; for defect detection and code classification CREAM also improves the performance of CodeBERT on the transformed test set by 8.3% and 1.9%, respectively. The results suggest that CREAM can eliminate the misleading impact of identifiers, enabling the models reliable to identifier renaming.

The robustness improvement on smaller datasets are more obvious. As shown in Table II, by analyzing the results on the datasets with different sizes for the function naming task, we find that CREAM achieves higher improvement on the smaller datasets. For example, CREAM boosts NCS by 116.9% and 83.7% on JavaScript and Ruby, respectively. This may be attributed to that neural models are prone to overfitting on datasets with small sizes [57], [58], thus the direct misleading information is prominent. Our proposed

CREAM can render the models less affected by identifier names especially on small datasets.

B. RQ2: Performance Evaluation

In this section, we evaluate the accuracy of CREAM on the code comprehension tasks. From Table II-IV, we observe that CREAM improves the performance of conventional models in most cases.

Function Naming. As shown in Table II, we find that CREAM improves the performance of each basic model in a vast majority of cases. Specifically, the average improvement of CREAM over CodeNN and NCS is 0.4% and 0.5%, respectively, regarding the F1 score. Although the averaged F1 score for CodeBERT+CREAM drops slightly, the performance on most languages still increases. The results show the effectiveness of CREAM on function naming.

Defect Detection. As shown in Table III, we can observe that CREAM improves the accuracy of all the basic models with an average improvement of 1.4%. Specifically, CodeBERT+CREAM and Devign+CREAM outperform their corresponding baselines by 0.9% and 1.7%, respectively, which indicates that CREAM can facilitate conventional models to capture the patterns of vulnerable code snippets.

Code Classification. As shown in Table IV, we can observe a consistent improvement of CREAM on different basic models. For example, although the performance of ASTNN and CodeBERT are strong enough, i.e., achieving 98.04% and 97.96% accuracy, respectively, CREAM can further boost them by 0.1% and 0.3%, respectively. This indicates that CREAM is also effective to comprehend the code functionality.

C. RQ3: Ablation Study

We further perform ablation studies to verify the effectiveness of two key stages in CREAM, i.e., multi-task learning and counterfactual inference. We select CodeBERT as the basic model, since it is used for evaluation on all the tasks. For function naming, we use the PHP dataset for evaluation.

Multi-task learning: L_f and L_t are introduced to distinguish the direct causal effect from the *total effect*. From

TABLE V: Ablation study. Best and second best results are marked in bold and underline respectively.

Approach	Function Naming (F1)		Defect Detection (accuracy)		Code Classification (accuracy)	
	Original	Transformed	Original	Transformed	Original	Transformed
CodeBERT	49.36	23.43	63.47	57.24	97.96	95.76
+CREAM	50.35	<u>35.80</u>	<u>64.05</u>	62.01	<u>98.28</u>	<u>97.56</u>
-w/o L_f	49.67	32.80	64.20	58.38	98.12	89.90
-w/o L_t	49.79	36.68	63.65	<u>61.42</u>	98.16	98.02
-w/o L_t and L_f	48.69	34.52	63.69	59.08	97.74	96.81
-w/o Counterfactual Inference	<u>50.16</u>	<u>31.26</u>	<u>63.87</u>	61.31	98.36	97.18

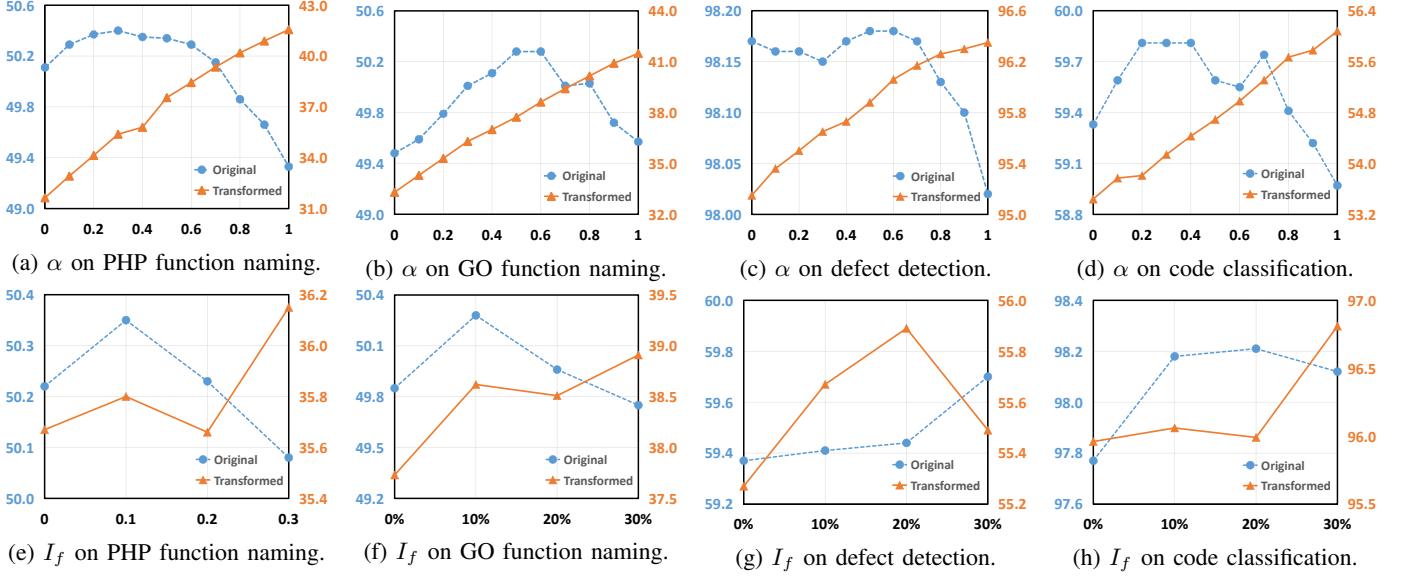


Fig. 6: Parameter analysis on α and I_{fusion} . I_f is the abbreviation of I_{fusion} . The vertical axis means the F1 score, accuracy and accuracy for function naming, defect detection and code classification respectively. The left and right vertical axes indicate results on the original and transformed dataset, respectively.

Table V, we can observe that models without them suffer from different degree of performance loss on the original test set or transformed test set. Specifically, removing L_f leads to a significant decrease on the transformed test set, with the decrease rate at 8.4%, 5.9% and 7.9% for function naming, defect detection and code classification, respectively; while without L_t , the framework performance drops consistently on the original test set. This indicates that removing L_f prevents the model from fully capturing the misleading information, which is harmful to model robustness; while removing L_t makes the model unable to distinguish $T \rightarrow R$ and $K \rightarrow R$ well, resulting in poorly exploiting the beneficial knowledge brought by $K \rightarrow R$. Moreover, removing both L_f and L_t leads to worse performance, e.g., a drop of 3.3% and 3.6% on the original and transformed test set of function naming, respectively.

Counterfactual inference: We validate the effectiveness of counterfactual inference by setting α in Equ. (11) to zero. As shown in the last row in Table V, without counterfactual inference, the framework's performance decreases on all tasks except a slightly improvement on the original test set for code classification. Specifically, the performance on the transformed test set drop by 12.7%, 1.1%, and 0.4% on function naming, defect detection and code classification, respectively. The

results show that the removal of misleading information by counterfactual inference improves the robustness of CREAM.

D. RQ4: Parameter Analysis

In this section, we analyze how the two key hyperparameters α and I_{fusion} affect the performance of CREAM. Due to the page limitation, in figure 6, we only present the experimental results with CodeBERT on the Go and PHP dataset, TextCNN and ASTNN as basic models for function naming, defect detection, and code classification, respectively. The results on other language and basic models are presented on our GitHub repository.⁵

The parameter α . As shown in Figure 6 (a), (b), (c) and (d), the model performance shows similar trend along with the increase of α on the original dataset for all the tasks. CREAM' performance first increases and achieves its peak, and then descends obviously with a larger α . The optimal α value is around 0.6 for the tasks. However, for the transformed dataset, the performance increases monotonically as α grows from 0 to 1. Recall that α in Equ. (17) is designed to control the degree of eliminating the misleading impact of identifiers, and a larger α will remove more misleading impact of the

⁵<https://github.com/ReliableCoding/CREAM>

identifiers. Since the misleading impact in the transformed dataset is more serious than that in the original dataset, a larger α in the transformed dataset is more appreciated. During the experimentation, to balance the performance of CREAM on both original and transformed datasets, we choose the value of α from the set $\{0.4, 0.5, 0.6, 0.7, 0.8\}$. Specifically, we test the model with α set from 0.4 to 0.8, and select the α with best robustness under the condition that the performance on the original dataset is not sacrificed too much.

The parameter I_{fusion} . We study the effect of I_{fusion} , as introduced in Section III-B2, by varying it from 0% to 30% of the total training iterations for the tasks. From Figure 6 (e), (f), (g) and (h), we can observe that involving I_{fusion} benefits the performance on both original and transformed datasets for all the tasks. However, for function naming, we also observe that the F1 score when I_{fusion} is set as 30% on the original dataset is lower than that when I_{fusion} is set as zero. This may be attributed to that models need more training epochs to well distinguish $T \rightarrow R$ and $K \rightarrow R$ for the complex generation task. In this work, we set I_{fusion} as 10% of the total training iterations due to the relatively better results on all tasks.

VI. DISCUSSION

A. Why Counterfactual Inference Helps?

In this section, we provide another Bayesian perspective to understand how CREAM eliminates the misleading impact of identifiers through counterfactual inference. We use $p_t(x, y)$ and $p_s(x, y)$ to denote the data distributions of the training set and test set, respectively. We focus on the common classification tasks [44], [47] which normalize classification scores with the softmax function. Based on Bayes' theorem, the posterior on the training and test set can be expressed as follows:

$$p_t(y|x) = \frac{p_t(y)p_t(x|y)}{\sum_y p_t(y)p_t(x|y)}, \quad (19)$$

$$p_s(y|x) = \frac{p_s(y)p_s(x|y)}{\sum_y p_s(y)p_s(x|y)}, \quad (20)$$

where $p(\cdot)$ is parameterized by a neural network. Our goal is to learn a set of parameters from training set, which is expected to estimate the posterior of test set well. Formally, given a sample x' in test set, we estimate its posterior by:

$$\bar{p}(y|x') = \frac{p_t(y)p_t(x'|y)}{\sum_y p_t(y)p_t(x'|y)} = \frac{e^{z'}}{\sum e^{z'}}, \quad (21)$$

where z' is the classification score for x' . Under random data split, we can assume that the prior on training and test set are the same, i.e. $p_t(y) = p_s(y)$. However, in practice, we cannot ensure that the likelihood on training set $p_t(x|y)$ is the same as that on test set $p_s(x|y)$ [59], [60]. For example, with respect to defect detection, if functions that contain the identifier “ss” are all labeled as vulnerable code in the training set, model will learn a higher vulnerable likelihood for the function with “ss”, i.e., $p_t(ss|vulnerable) > p_t(ss|invulnerable)$.

TABLE VI: Comparison results of attack success rates on attacking CodeBERT and CodeBERT+CREAM. MHM-NS denotes MHM attack with natural substitution [61].

Approach	Model	Defect detection	Code classification
ALERT	CodeBERT	53.97	46.43
	+CREAM	24.62 ($\downarrow 54.38\%$)	20.77 ($\downarrow 55.27\%$)
MHM-NS	CodeBERT	33.27	4.15
	+CREAM	17.32 ($\downarrow 48.21\%$)	1.21 ($\downarrow 70.84\%$)

Thus, conventional models that directly predict based on the likelihood learned on the training set may be misled, since the identifier “ss” is irrelevant to the vulnerability of the function. Different from conventional models, our framework adaptively estimates and subtracts the misleading impact of identifiers from the classification score in the inference stage:

$$\frac{e^{z'-f_{z'}}}{\sum_z e^{z'-f_{z'}}} = \frac{p_t(y)p_t(x'|y)\beta_y}{\sum_y p_t(y)p_t(x'|y)\beta_y} \quad (22)$$

where $f_{z'}$ is corresponding to $\alpha * R_{f,k^*,t^*}$ in Equ. (17). Here, the stronger the correlation between the identifiers in x' and y , the larger (smaller) the value of $f_{z'}$ (β_y) will be. This indicates that CREAM eliminates the misleading impact of identifiers by rectifying the incorrect estimation of likelihood learned from the training set. In this work, we mainly focus on the misleading information caused by identifiers, and will explore other potential misleading sources in future work.

B. Performance under adversarial attacks

In this section, we follow previous work [61]–[63] and further evaluate the robustness improvement of CREAM by adversarial attack. Specifically, we experiment with CodeBERT on defect detection and code classification since it shows the best performance on both tasks. For adversarial attack methods, we select two state-of-the-art black-box methods ALERT [61] and MHM [16] with natural substitution [61]. We evaluate the robustness of model by the widely-used Attack Success Rates (ASR) metric [61], [63], [64], which measures the fraction of samples that can be attacked among all of the test samples. A higher ASR indicates that a model is more vulnerable to adversarial attack. From Table VI, we can observe that CREAM consistently improves the robustness of CodeBERT on both tasks under both attack approaches. For example, CREAM reduces 54.38% and 55.27% possibility of being attacked by ALERT on defect detection and code classification, respectively. This indicates that CREAM can also effectively improve the robustness of neural code comprehension models under adversarial attacks. We will experiment with more basic models and adversarial attack methods in our future work.

C. Threats to Validity

We identify three main threats to validity of our study:

- 1) **The selection of code comprehension tasks.** In this work, we select three popular code comprehension tasks to evaluate CREAM, including function naming, defect detection and code classification. Although CREAM

shows superior performance on these tasks, other tasks such as code search [4], [5] and code summarization [6], [7], [65] are also important and not involved in our experiment. In the future, we will validate CREAM on more code comprehension tasks.

- 2) **The selection of basic models.** For each task, we select at least three basic models to validate the effectiveness of CREAM. The selected basic models are representative of the corresponding task. To comprehensively evaluate the performance of CREAM, more basic models should be considered. In the future, we will experiment with more basic models to evaluate the generality of CREAM.
- 3) **The selection of datasets.** For each task, we select one popular dataset for evaluation. However, there are other datasets such as [66] for function naming. In the future, we will conduct experiments on more datasets.
- 4) **Comparison to ensemble techniques.** CREAM aggregates the prediction from three branches with different inputs, which can also be framed as ensemble learning. There is thus a threat that some simple ensemble learning methods can also improve the model’s performance. In the future, we will compare CREAM with other ensemble techniques to validate the benefits of counterfactual reasoning.

VII. RELATED WORK

A. Code Comprehension

In this section, we focus on deep-learning-based methods on three tasks that are covered in our work including function naming, defect detection and code classification. Besides, the related work on pre-trained models for code are also discussed.

Function Naming: Alon et al. [8] present Code2seq that represents the code snippets by sampling certain paths from the ASTs. Another work proposed by Zügner et al. [9] focuses on multilingual code summarization and proposes to build upon language-agnostic features such as source code and AST-based features. A recent work [67] propose to encodes tree paths into transformer.

Defect Detection: Russell et al. [50] empirically evaluate the ML techniques on defect detection and find that TextCNN with an ensemble tree algorithm achieves the best performance. Another work [49] proposes the first deep learning-based vulnerability detection system VulDeePecker. Devign [11] is proposed to learn the various vulnerability characteristics with a composite code property graph and graph neural network.

Code classification: TBCNN [44] is a classical code classification model which captures structural information of the AST with a tree-based convolutional neural network. ASTNN [11] learns the code representation by splitting the large AST into a sequence of small statement trees. Another recent work [68] propose to capture the tree structure of code with a capsule network.

Pre-trained models for code: Recently, a number of pre-trained models for source code have been proposed [45], [52], [69]. CodeBERT [45] is an encoder-only pre-trained model

based on Masked language modeling and replaced token detection. GraphCodeBERT [69] further leverage code structure information by data flow graph. Another recent work [70] proposes a sequence-to-sequence pre-trained model with the encoder-decoder architecture.

B. Causal Inference

Causal inference has attracted increasing attention in fields including computer vision [29], [71], natural language processing [72], [73] and recommendation [30], [39]. The general purpose of causal inference is to help model pursue causal effect rather than correlation effect. Niu et al. [29] propose a counterfactual framework to remove the language bias in visual question answering. In recommendation, [39] and [30] also employ similar methods to eliminate the popularity bias. Some works [74], [75] also consider to build the causal graph from data generation view and remove the confounder with back-door adjustment. In text classification, some works [72], [76] focus on alleviate the spurious correlation by generating counterfactual samples. Different from the above studies, we are devoted to extracting and eliminating the misleading impact of identifiers in neural code models. To the best of our knowledge, we are the first to introduce the causal inference into neural code comprehension.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we present CREAM, a counterfactual reasoning-based framework to eliminate the misleading impact of identifiers in neural code comprehension. CREAM captures the misleading information in the training stage through multi-task learning and reduces it by counterfactual inference in the inference stage. CREAM is flexible and easy to be applied to various tasks and basic models. The evaluation on three popular tasks demonstrates the effectiveness of CREAM on original test sets and its robustness to identifier renaming. In the future, we will explore to apply more causal inference techniques to solve the challenges in code intelligence tasks.

Data availability: The implementation repository of this work is publicly available at <https://github.com/ReliableCoding/CREAM>.

REFERENCES

- [1] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. IEEE / ACM, 2019, pp. 783–794.
- [2] F. Liu, G. Li, Y. Zhao, and Z. Jin, “Multi-task learning based pre-trained language model for code completion,” in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 2020, pp. 473–485.
- [3] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, “Retrieve and refine: Exemplar-based neural comment generation,” in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 2020, pp. 349–360.
- [4] W. Gu, Z. Li, C. Gao, C. Wang, H. Zhang, Z. Xu, and M. R. Lyu, “Cradle: Deep code retrieval based on semantic dependency learning,” *Neural Networks*, vol. 141, pp. 385–394, 2021.
- [5] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, “Improving code search with co-attentive representation learning,” in *ICPC ’20: 28th International Conference on Program Comprehension*. ACM, 2020, pp. 196–207.

- [6] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Summarizing source code using a neural attention model,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016*. The Association for Computer Linguistics, 2016.
- [7] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, F. Khomh, C. K. Roy, and J. Siegmund, Eds. ACM, 2018, pp. 200–210.
- [8] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” in *7th International Conference on Learning Representations, ICLR 2019*. OpenReview.net, 2019.
- [9] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, “Language-agnostic representation learning of source code from structure and context,” in *9th International Conference on Learning Representations, ICLR 2021*. OpenReview.net, 2021.
- [10] Y. Li, S. Wang, and T. N. Nguyen, “A context-based automated approach for method name consistency checking and suggestion,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*. IEEE, 2021, pp. 574–586.
- [11] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, 2019, pp. 10197–10207.
- [12] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye, “Interpreting deep learning-based vulnerability detector predictions based on heuristic searching,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 23:1–23:31, 2021.
- [13] Y. Li, S. Wang, and T. N. Nguyen, “Vulnerability detection with fine-grained interpretations,” in *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 2021, pp. 292–303.
- [14] V. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, “Regvd: Revisiting graph neural networks for vulnerability detection,” in *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 2022, pp. 178–182.
- [15] M. R. I. Rabin, N. D. Q. 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, p. 106552, 2021.
- [16] 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 *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*. AAAI Press, 2020, pp. 1169–1176.
- [17] N. Yefet, U. Alon, and E. Yahav, “Adversarial examples for models of code,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 162:1–162:30, 2020.
- [18] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, “Towards robustness of deep program processing models—detection, estimation and enhancement,” *ACM Transactions on Software Engineering and Methodology*, 2022.
- [19] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. W. Reps, “Semantic robustness of models of source code,” *CoRR*, vol. abs/2002.03043, 2020.
- [20] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, “A transformer-based approach for source code summarization,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020*. Association for Computational Linguistics, 2020, pp. 4998–5007.
- [21] N. Chirkova and S. Troshin, “Empirical study of transformers for source code,” in *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021, pp. 703–715.
- [22] A. Mastropaoletti, S. Scalabrino, N. Cooper, D. Nader-Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 336–347.
- [23] D. Castelvecchi, “Can we open the black box of ai?” *Nature News*, vol. 538, no. 7623, p. 20, 2016.
- [24] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, “Green ai,” *Communications of the ACM*, vol. 63, no. 12, pp. 54–63, 2020.
- [25] X. Zhu and A. B. Goldberg, “Introduction to semi-supervised learning,” *Synthesis lectures on artificial intelligence and machine learning*, vol. 3, no. 1, pp. 1–130, 2009.
- [26] J. Pearl, *Causality*. Cambridge university press, 2009.
- [27] D. P. MacKinnon, A. J. Fairchild, and M. S. Fritz, “Mediation analysis,” *Annu. Rev. Psychol.*, vol. 58, pp. 593–614, 2007.
- [28] L. Keele, “The statistics of causal inference: A view from political methodology,” *Political Analysis*, vol. 23, no. 3, pp. 313–335, 2015.
- [29] Y. Niu, K. Tang, H. Zhang, Z. Lu, X. Hua, and J. Wen, “Counterfactual VQA: A cause-effect look at language bias,” in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021*. Computer Vision Foundation / IEEE, 2021, pp. 12700–12710.
- [30] T. Wei, F. Feng, J. Chen, Z. Wu, J. Yi, and X. He, “Model-agnostic counterfactual reasoning for eliminating popularity bias in recommender system,” in *KDD ’21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, 2021, pp. 1791–1800.
- [31] J. Pearl and D. Mackenzie, *The book of why: the new science of cause and effect*. Basic books, 2018.
- [32] J. Pearl, “Graphs, causality, and structural equation models,” *Sociological Methods & Research*, vol. 27, no. 2, pp. 226–284, 1998.
- [33] M. Glymour, J. Pearl, and N. P. Jewell, *Causal inference in statistics: A primer*. John Wiley & Sons, 2016.
- [34] T. J. VanderWeele, “A three-way decomposition of a total effect into direct, indirect, and interactive effects,” *Epidemiology (Cambridge, Mass.)*, vol. 24, no. 2, p. 224, 2013.
- [35] G. E. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” *CoRR*, vol. abs/1503.02531, 2015.
- [36] A. K. Menon, S. Jayasumana, A. S. Rawat, H. Jain, A. Veit, and S. Kumar, “Long-tail learning via logit adjustment,” in *9th International Conference on Learning Representations, ICLR 2021*. OpenReview.net, 2021.
- [37] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, 2017, pp. 5998–6008.
- [39] W. Wang, F. Feng, X. He, H. Zhang, and T. Chua, “Clicks can be cheating: Counterfactual recommendation for mitigating clickbait issue,” in *SIGIR ’21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2021, pp. 1288–1297.
- [40] K. Cao, C. Wei, A. Gaidon, N. Aréchiga, and T. Ma, “Learning imbalanced datasets with label-distribution-aware margin loss,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, 2019, pp. 1565–1576.
- [41] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, “Suggesting natural method names to check name consistencies,” in *ICSE ’20: 42nd International Conference on Software Engineering*. ACM, 2020, pp. 1372–1384.
- [42] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin, “Learning to recommend method names with global context,” *CoRR*, vol. abs/2201.10705, 2022.
- [43] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.
- [44] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press, 2016, pp. 1287–1293.
- [45] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “Codebert: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.
- [46] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” *CoRR*, vol. abs/2102.04664, 2021.

- [47] Y. Kim, “Convolutional neural networks for sentence classification,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*. ACL, 2014, pp. 1746–1751.
- [48] Z. Yang, D. Yang, C. Dyer, X. He, A. J. Smola, and E. H. Hovy, “Hierarchical attention networks for document classification,” in *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. The Association for Computational Linguistics, 2016, pp. 1480–1489.
- [49] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection,” in *25th Annual Network and Distributed System Security Symposium, NDSS 2018*. The Internet Society, 2018.
- [50] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, “Automated vulnerability detection in source code using deep representation learning,” in *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018*. IEEE, 2018, pp. 757–762.
- [51] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” *CoRR*, vol. abs/1909.09436, 2019.
- [52] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*. Association for Computational Linguistics, 2021, pp. 8696–8708.
- [53] N. D. Q. Bui, Y. Yu, and L. Jiang, “Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations,” in *SIGIR ’21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2021, pp. 511–521.
- [54] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, “Challenging machine learning-based clone detectors via semantic-preserving code transformations,” *CoRR*, vol. abs/2111.10793, 2021.
- [55] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, “An extensive study on pre-trained models for program understanding and generation,” in *ISSTA ’22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 39–51.
- [56] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet,” *IEEE Transactions on Software Engineering*, 2021.
- [57] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, “Generalizing from a few examples: A survey on few-shot learning,” *ACM Comput. Surv.*, vol. 53, no. 3, pp. 63:1–63:34, 2020.
- [58] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, “Understanding deep learning requires rethinking generalization,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.
- [59] J. Gama, I. Zliobaite, A. Bifet, M. Pechenizkiy, and A. Bouchachia, “A survey on concept drift adaptation,” *ACM Comput. Surv.*, vol. 46, no. 4, pp. 44:1–44:37, 2014.
- [60] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, “Learning under concept drift: A review,” *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 12, pp. 2346–2363, 2019.
- [61] Z. Yang, J. Shi, J. He, and D. Lo, “Natural attack for pre-trained models of code,” in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1482–1493.
- [62] Z. Li, Y. Li, T. Li, M. Du, B. Wu, Y. Cao, X. Xie, Y. Li, and Y. Liu, “Unveiling project-specific bias in neural code models,” *CoRR*, vol. abs/2201.07381, 2022.
- [63] Z. Li, Q. G. Chen, C. Chen, Y. Zou, and S. Xu, “Ropgen: Towards robust code authorship attribution via automatic coding style transformation,” in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1906–1918.
- [64] Y. Dong, Q. Fu, X. Yang, T. Pang, H. Su, Z. Xiao, and J. Zhu, “Benchmarking adversarial robustness on image classification,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. Computer Vision Foundation / IEEE, 2020, pp. 318–328.
- [65] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, X. Xia, and M. R. Lyu, “Code structure guided transformer for source code summarization,” *ACM Trans. Softw. Eng. Methodol.*, 2022.
- [66] M. Allamanis, H. Peng, and C. Sutton, “A convolutional attention network for extreme summarization of source code,” in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, ser. JMLR Workshop and Conference Proceedings, vol. 48. JMLR.org, 2016, pp. 2091–2100.
- [67] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin, “Integrating tree path in transformer for code representation,” *Advances in Neural Information Processing Systems*, vol. 34, 2021.
- [68] N. D. Q. Bui, Y. Yu, and L. Jiang, “Treecaps: Tree-based capsule networks for source code processing,” in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*. AAAI Press, 2021, pp. 30–38.
- [69] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” in *9th International Conference on Learning Representations, ICLR 2021*. OpenReview.net, 2021.
- [70] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, “Spt-code: Sequence-to-sequence pre-training for learning source code representations,” *CoRR*, vol. abs/2201.01549, 2022.
- [71] T. Wang, C. Zhou, Q. Sun, and H. Zhang, “Causal attention for unbiased visual recognition,” in *2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021*. IEEE, 2021, pp. 3071–3080.
- [72] Z. Wang and A. Culotta, “Robustness to spurious correlations in text classification via automatically generated counterfactuals,” in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*. AAAI Press, 2021, pp. 14 024–14 031.
- [73] C. Qian, F. Feng, L. Wen, C. Ma, and P. Xie, “Counterfactual inference for text classification debiasing,” in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021*. Association for Computational Linguistics, 2021, pp. 5434–5445.
- [74] Y. Zhang, F. Feng, X. He, T. Wei, C. Song, G. Ling, and Y. Zhang, “Causal intervention for leveraging popularity bias in recommendation,” in *SIGIR ’21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2021, pp. 11–20.
- [75] X. Yang, F. Feng, W. Ji, M. Wang, and T. Chua, “Deconfounded video moment retrieval with causal intervention,” in *SIGIR ’21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2021, pp. 1–10.
- [76] L. Yang, J. Li, P. Cunningham, Y. Zhang, B. Smyth, and R. Dong, “Exploring the efficacy of automatically generated counterfactuals for sentiment analysis,” in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021*. Association for Computational Linguistics, 2021, pp. 306–316.