CodeFort: Robust Training for Code Generation Models

Yuhao Zhang[1*], Shiqi Wang[1], Haifeng Qian[1], Zijian Wang[1], Mingyue Shang[1], Linbo Liu[1], Sanjay Krishna Gouda[1], Baishakhi Ray[1], Murali Krishna Ramanathan[1], Xiaofei Ma[1], Anoop Deoras[1]

[1]AWS AI Labs,

Correspondence: yhzhangh@amazon.com

Abstract

Code generation models are not robust to small perturbations, which often lead to incorrect generations and significantly degrade the performance of these models. Although improving the robustness of code generation models is crucial to enhancing user experience in real-world applications, existing research efforts do not address this issue. To fill this gap, we propose CodeFort, a framework to improve the robustness of code generation models, generalizing a large variety of code perturbations to enrich the training data and enabling various robust training strategies, mixing data augmentation, batch augmentation, adversarial logits pairing, and contrastive learning, all carefully designed to support high-throughput training. Extensive evaluations show that we increase the average robust pass rates of baseline CodeGen models from 14.79 to 21.74. Notably, we decrease the robustness drop rate from 95.02% to 54.95% against code-syntax perturbations.

1 Introduction

Code generation models (Li et al., 2023; Nijkamp et al., 2023a; Fried et al., 2023; Luo et al., 2023; Rozière et al., 2023) have demonstrated impressive performance in generating code from natural language descriptions, completing sections of code, and even tackling complex coding contest challenges. These models can potentially assist software engineers and increase their productivity.

However, code generation models are not robust to minor perturbations in the input prompts (e.g., inserting whitespaces/typos in docstrings or substituting variable names in code), i.e., they often generate incorrect outputs, thus significantly degrading their impressive performance on nominal prompts and hurting user experience when deployed in real-world applications (Wang et al., 2023b). Figure 1 shows that the performance of the state-of-the-art public code models (Nijkamp et al., 2023b; Li et al., 2023; Luo et al., 2023) significantly declines under semantic-preserving program transformations, particularly under code-syntax perturbations. Thus, it is crucial to improve the robustness of models before they are universally deployed.

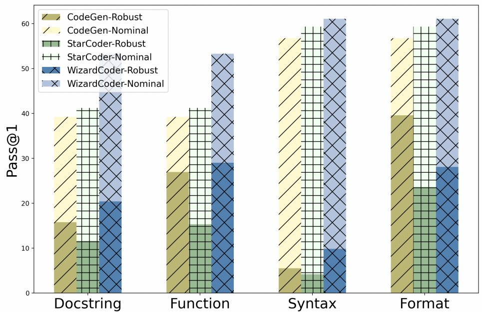

Figure 1: Performance drop of the state-of-the-art public code models on four classes of code perturbations. (See Table 4 in the Appendix for details.)

Despite extensive research efforts to improve the robustness of code-related tasks, beyond code generation, such as vulnerability prediction, clone detection, and code summarization, existing work has not tackled two unique challenges of improving the robustness of code generation models, primarily trained using casual language modeling (CLM).

Challenge 1: Distinct Robustness Definition Unlike traditional classification tasks like vulnerability detection, where models produce a single classification, code generation models generate sequences, leading to a shift in the definition of robustness for certain perturbations. In code generation, model robustness is defined by generating a *coherent* output given an input perturbation. In contrast, in classification tasks, models are expected to maintain the *same* classification before and after perturbation. For instance, if a variable `i` is renamed to `b` in the input prompt (as illustrated in Figure 2d), a robust code generation model should generate completions with the variable `b` instead of the original variable `i`. This shift necessitates a new

[*]Work done when the author was an intern at Amazon.

Findings of the Association for Computational Linguistics: EMNLP 2024, pages 5262–5277
November 12-16, 2024 ©2024 Association for Computational Linguistics

category of perturbations for code generation models and corresponding robust training approaches to tackle this new category.

Challenge 2: Designing Robust Training Approaches As code perturbations can insert dead code or typos into training data, directly training using data augmentation could adversely affect the model performance, leading to issues like generating dead code or typos. Furthermore, applying more deliberate robust training approaches such as adversarial logits pairing (ALP) (Kannan et al., 2018) and contrastive learning (CL) to CLM presents unique challenges. ALP, designed for single-class classification, requires careful alignment between original and perturbed sequences, complicated by potential differences in sequence lengths. Although CL has demonstrated efficacy in improving the robustness of code representations in masked language modeling (MLM) (Devlin et al., 2019), its applicability to improving the robustness of code generation models remains unexplored.

To tackle the above two challenges and improve the robustness of code generation models, we introduce a structured definition of code perturbations (Section 3) and design a novel robust training framework named CodeFort (Section 4).

To address Challenge 1, we classify existing code perturbations into two categories: *context-free* and *context-sensitive*, based on the formal definition of code perturbations provided in Section 3. Context-free perturbations, such as the docstring perturbation in Figure 2b, follow the traditional notions of robustness, whereas context-sensitive perturbations, such as the code-syntax perturbation in Figure 2d, specific the distinct robustness definition highlighted in Challenge 1. The distinction of these two categories allows CodeFort to employ different robust training methods according to each category.

To address Challenge 2, CodeFort employs example-level and sequence-level pairing to enrich the training set. These two pairing levels allow 1) a masking mechanism to mask unnatural perturbed tokens and 2) a careful alignment between the original and perturbed token sequences, addressing the crucial challenge of applying ALP and CL to CLM.

We utilize CodeFort to extensively evaluate various strategies, mixing data augmentation, batch augmentation, ALP, and CL. Our approach, combining batch augmentation with the masking mechanism, ALP, and ALPD, significantly enhances the model robustness and surpasses the sub-optimal results of data augmentation. Notably, our approach significantly improves the robustness under code-syntax perturbations, the type of perturbation that hurts the model robustness the most, as shown in Figure 1. Our ablation studies show that ContraSeq, the CL objective used in previous work for MLM, has negligible robustness improvements on CLM.

We summarize our contributions: 1) a framework, CodeFort, for improving the robustness of code generation models trained by CLM, 2) designs of robust training approaches, data augmentation, batch augmentation, ALP, and CL tailored to CLM, 3) an extensive evaluation of different robust training approaches, 4) a perturbed training set for future studies on the robustness of code generation models, and 5) a surprising finding that the ContraSeq CL objective, which is known to be beneficial for improving robustness of other code related tasks, has negligible robustness improvements on CLM.

2 Related Work

Adversarial Attacks on Code-Related Tasks Numerous adversarial attacks (Henkel et al., 2022; Zhang et al., 2020; Yefet et al., 2020; Jha and Reddy, 2023; Srikant et al., 2021; Anand et al., 2021; Gao et al., 2023) have targeted encoder-decoder models in code-related tasks, including classification (e.g., vulnerability prediction) and generation (e.g., code summarization). Key methods include CODA (Tian et al., 2023), which exploits syntactic differences for adversarial example generation; CARROT (Zhang et al., 2022), employing a lightweight hill climbing for optimization in attacks; and ALERT (Yang et al., 2022), which creates naturalness-aware attacks using pre-trained models. Unlike these approaches, we focus on improving the robustness of *code generation* models trained using *casual language modeling* (CLM). We assess our approaches' effectiveness in code generation through ReCode (Wang et al., 2023b), a benchmark for evaluating robustness via semantic-preserving program transformations.

Robust Training on Code-Related Tasks Existing work typically enhances model robustness through data augmentation and adversarial training (Madry et al., 2018). Bielik and Vechev (2020) refine model representations by feeding only pertinent program parts to the model; Suneja et al.

```
def largest_divisor(n: int) -> int:
    """ For a given number n, find the largest number that
↪divides n evenly, smaller than n
    >>> largest_divisor(15)
    5
    """
===
    for i in reversed(range(n)):
        if n % i == 0:
            return i
```

(a) An original problem in HumanEval. === separates the prompt and the ground-truth completion.

```
def largestDivisor(n: int) -> int:
    """ For a given number n, find the largest number that
↪separate n evenly, modest than n
    >>> largestDivisor(15)
    5
    """
===
    for i in reversed(range(n)):
        if n % i == 0:
            return i
```

(b) A perturbed version of Figure 2a by a function-name and docstring perturbation.

```
def largest_divisor(n: int) -> int:
    """ For a given number n, find the largest number that
↪divides n evenly, smaller than n
    >>> largest_divisor(15)
    5
    """
    for i in reversed(range(n)):
===
        if n % i == 0:
            return i
```

(c) A HumanEval problem includes the first half of the original completion.

```
def largest_divisor(n: int) -> int:
    """ For a given number n, find the largest number that
↪divides n evenly, smaller than n
    >>> largest_divisor(15)
    5
    """
    for b \
  in reversed(range(n)):
===
        if n % b == 0:
            return b
```

(d) A perturbed version of Figure 2c by a code-syntax and code-format perturbation.

Figure 2: HumanEval problems under different code perturbations. To achieve a more compact illustration, we merge two code perturbations in one example.

(2023) use curriculum learning and data augmentation with simplified programs. They all tend to improve robustness in classification tasks. Unlike these, our focus is on code generation robustness.

While Zhou et al. (2022) propose random input token masking to lessen dependence on non-robust features, our method selectively masks perturbed tokens during loss calculation to avoid the model generating unnatural perturbations.

In contrast to ContraCode (Jain et al., 2021) and ContraBERT (Liu et al., 2023), which apply contrastive learning (CL) to classification and code translation tasks by improving robustness in masked language modeling, we focus on the efficacy of CL in decoder-only code generation models. Furthermore, directly applying the CL objective of ContraBERT and ContraCode on sequence representations may not cater to CLM, which involves discriminating representations at a finer level than sequence representations. Notably, this adoption shows negligible robustness improvement on CLM code models (Section 5.3). Thus, we need to design novel CL objectives tailored to finer granularities.

Although ContraCLM (Jain et al., 2023) enhances the discrimination of CLM's representations, it does not target robustness improvement.

3 Problem Definition

We address the robustness challenge in a code generation model f trained using Causal Language Modeling (CLM). CLM predicts the next token in a sequence, and the model can only attend to tokens on the left. Formally, given a sequence of tokens $\mathbf{x} = x_1, \ldots, x_n$, the generation model f captures $p_f(\cdot \mid \mathbf{x}_{:i})$, representing the conditional probabilities of the i-th token given the preceding tokens $\mathbf{x}_{:i} = x_1, \ldots, x_{i-1}$. The model is trained on a dataset $D = \{\mathbf{x}^j\}_{j=1}^m$ using cross-entropy loss $\mathcal{L}_{\text{CLM}}(\mathbf{x}) = -\sum_{i=1}^n \log p_f(x_i \mid \mathbf{x}_{:i})$.

Utilizing a given decoding strategy, such as greedy or temperature sampling (Holtzman et al., 2020), the generation model f produces a sequence of tokens by iteratively predicting the next tokens until a specified stop criterion is reached. We denote $f(\mathbf{x}_{:i}) = \hat{\mathbf{x}}_{i:}$ as the generated token sequence by f. The terms *prompt*, *completion*, and *ground truth* refer to the input $\mathbf{x}_{:i}$, the output $\hat{\mathbf{x}}_{i:}$, and the original completion $\mathbf{x}_{i:} = x_i, \ldots, x_n$, respectively.

In code generation, the token sequence $\mathbf{x}$ represents a code snippet. Figure 2a shows a problem in HumanEval (Chen et al., 2021). Each prompt $\mathbf{x}_{:i}$ in a problem contains a function signature and a corresponding docstring description, and each ground-truth completion $\mathbf{x}_{i:}$ is the correct function implementation. Given a completion $\hat{\mathbf{x}}_{i:}$ generated by a model f, if the completed function $\mathbf{x}_{:i} + \hat{\mathbf{x}}_{i:}$ passes all the hidden test cases, the completion is deemed correct, denoted as $\text{Cor}(\mathbf{x}_{:i} + \hat{\mathbf{x}}_{i:}) = \textit{true}$. Otherwise, if any of the tests fail, $\text{Cor}(\mathbf{x}_{:i} + \hat{\mathbf{x}}_{i:}) = \textit{false}$.

3.1 Code Perturbations

ReCode (Wang et al., 2023b) is a comprehensive robustness evaluation benchmark for code generation models containing semantic-preserving code perturbations across four classes: docstring, function-name, code-syntax, and code-format perturbations.

We present examples from these four classes and we encourage readers to refer to the original paper for more detailed descriptions. Table 5 in the Appendix presents a complete list of code perturbations used in this paper.

Docstring Perturbations rewrite natural language in docstrings and comments, including edits like substituting synonyms (Figure 2b). *Function-Name Perturbations* refactor some function names, e.g., changing from snake_case to camelCase (Figure 2b). *Code-Syntax Perturbations* apply perturbations related to code syntax, involving changes like renaming variables (Figure 2d). *Code-Format Perturbations* change the code snippets' format, e.g., splitting a line into two (Figure 2d).

A code perturbation is a collection of string transformations, denoted as $\pi = \{T_1, T_2, \ldots\}$. Each transformation $T : \mathcal{X} \mapsto \mathcal{X}$ operates on a token sequence from the input domain $\mathcal{X}$, altering them to produce a perturbed sequence. Within π, each transformation specifies different positions and replacements for perturbation.

Example 3.1. *The VarRenamer perturbation, shown in Figure 2c, is a code perturbation π. It contains an infinite set of string transformations that specify 1) which variable name to change and 2) the new variable name. The former contains two choices:* `n` *and* `i`. *And the latter contains infinite choices of valid variable names.*

We introduce two categories, *context-free* and *context-sensitive* perturbations, which serve as a high-level interface for robust training. We formalize the distinctive characteristics of these two categories in the following sections.

3.1.1 Context-Free Perturbations

A code perturbation π is a *context-free* perturbation if all perturbed prompts generated by π should not affect the ground-truth completion. Formally, for all $T \in \pi$, the concatenation of the prompt perturbed by T and the ground-truth completion remains a correct function:

$$\forall T \in \pi, \mathrm{Cor}(T(\mathbf{x}_{:i}) + \mathbf{x}_{i:}) \tag{1}$$

Example 3.2. *In Figure 2b, the SynonymSubstitution perturbation in the docstring will not affect the ground-truth completion.*

3.1.2 Context-Sensitive Perturbations

A code perturbation π is a *context-sensitive* perturbation if any perturbed prompt generated by π results in *coherent* changes to the ground-truth completion. Formally, for all $T \in \pi$, the concatenation of the prompt perturbed by T and its ground-truth completion perturbed correspondingly is a correct function, while the concatenation of the perturbed prompt and the original completion is not.

$$\forall T \in \pi, \mathrm{Cor}(T(\mathbf{x}_{:i}) + T(\mathbf{x}_{i:})) \wedge \neg\mathrm{Cor}(T(\mathbf{x}_{:i}) + \mathbf{x}_{i:}) \tag{2}$$

Example 3.3. *In Figure 2d, the VarRenamer perturbation requires the ground-truth completion to change* ***coherently*** *because all the variable* `i` *should be renamed to* `b`.

3.2 Robustness of Code Generation Models

To define model robustness, we say a model f is robust to a perturbation π, if

$$\forall T \in \pi, \mathrm{Cor}(T(\mathbf{x}_{:i}) + f(T(\mathbf{x}_{:i}))) \tag{3}$$

The robustness against context-free perturbations is similar to the traditional robustness definition, in which the perturbation should not change the model results (Eq 1). However, the robustness against context-sensitive perturbations differs from the traditional definition, as the context-sensitive robustness requires the model output to change coherently with the perturbed prompt (Eq 2).

4 CodeFort

CodeFort enhances the training set with a paired dataset generation method (Section 4.1). Section 4.2 outlines our robust training strategies.

4.1 Paired Dataset Generation Method

The paired dataset generation method offers two levels of pairings: *example-level* and *sequence-level*. Example-level pairing matches each original training example with its perturbed counterpart. Sequence-level pairing provides more detail by matching each token segment from the original example to its equivalent in the perturbed example. These pairing mechanisms are essential for the robust training strategies discussed in Section 4.2.

Example-level Pairing Given a training set $D = \{\mathbf{x}^j\}_{j=1}^m$, where each training example is a code snippet, the paired dataset generation method returns a set of paired of training samples $\{(\mathbf{x}^j, \tilde{\mathbf{x}}^j)\}$, where $\tilde{\mathbf{x}}^j$ is the code snippet perturbed by some code perturbations. To obtain $\tilde{\mathbf{x}}^j$, we first randomly choose t code perturbations $\{\pi_1, \ldots, \pi_t\}$ in ReCode. Then, we randomly choose one string transformation from each code perturbation and apply it to the original code snippet $\mathbf{x}$,

$$\tilde{\mathbf{x}} = T_t(T_{t-1}(\ldots T_1(\mathbf{x})\ldots)), \quad T_i \in \pi_i, \forall 1 \le i \le t. \quad (4)$$

Sequence-level Pairing Given a pair of code snippets, $(\mathbf{x}, \tilde{\mathbf{x}})$, CodeFort further provides finer-granularity paring for this pair.

Example 4.1. *Consider following pairs of tokens,*

```
Index:       0 1 2 3 4 5 6 7 8 9
Original:    A C D E F G H C I
Perturbed:   A B X D E F Y X Z Z
```

In this example, the code perturbations perform two context-free perturbations, insert "B" and substitute "G H" with "Y", and two context-sensitive perturbations, substitute all "C" with "X" and substitute all "I" with "Z Z".

To create sequence-level pairing, we introduce a mask sequence $\mathbf{m}$ ($\tilde{\mathbf{m}}$) for the original sequence $\mathbf{x}$ (and the perturbed one $\tilde{\mathbf{x}}$, respectively). Each mask value indicates which kind of perturbation is applied to the corresponding token, U for unperturbed, F for context-free, and S for context-sensitive.

Example 4.2. *We show two masks of Example 4.1.*

```
Index:           0 1 2 3 4 5 6 7 8 9
Original Mask:   U S U U U F F S S
Perturbed Mask:  U F S U U U F S S S
```

This two-level pairing design is the key to enabling some of the robust training approaches, which will be introduced subsequently.

4.2 Designing Robust Training Approaches

This section introduces four robust training approaches in CodeFort.

4.2.1 Data Augmentation

Data augmentation is a widely used approach to improve the robustness of machine learning models. A common practice is replacing a certain portion, denoted as p, of the original training examples with their perturbed counterparts in each training batch. Formally, for a training batch $\{\mathbf{x}^j\}_{j=1}^b$ and its paired perturbed batch $\{\tilde{\mathbf{x}}^j\}_{j=1}^b$, the objective function is expressed as follows,

$$\mathcal{L}_{\mathrm{DA}} = \sum_{j=1}^{b} a_j \mathcal{L}_{\mathrm{CLM}}(\tilde{\mathbf{x}}^j) + (1 - a_j)\mathcal{L}_{\mathrm{CLM}}(\mathbf{x}^j), \quad (5)$$

where $a_j \overset{\text{i.i.d}}{\sim} \text{Bernoulli}(p)$ is a Bernoulli variable indicating whether the j-th training example will be perturbed or not.

Masking Unnatural Perturbed Tokens Some context-free perturbations introduce unnatural tokens, such as DeadCode Insertion adding an artificial code segment and ButterFingers introducing typos. Referring back to the robustness property in Eq 3, our goal is for the model to learn to respond to these perturbations rather than to generate them. Learning to generate these unnatural perturbed tokens could adversely affect the original model performance, leading to issues like generating dead code or typos. We propose masking the CLM loss of these unnatural perturbed tokens to address these issues. We define the CLM loss for the example $\mathbf{x}$ after masking out the unnatural perturbed tokens

$$\mathcal{L}_{\mathrm{CLM}}(\mathbf{x}, \mathbf{m}) = -\sum_{i=1}^{|\mathbf{x}|} \mathbb{1}_{\{m_i \ne \mathrm{F}\}} \log p_f(x_i \mid \mathbf{x}_{:i}),$$

where $m_i \ne$ F means that the i-th token is not perturbed by a context-free perturbation (see Example 4.2). We design the masked data augmentation loss $\mathcal{L}_{\mathrm{MDA}}$ by replacing the term $\mathcal{L}_{\mathrm{CLM}}(\tilde{\mathbf{x}})$ in Eq 5 with the masked loss $\mathcal{L}_{\mathrm{CLM}}(\tilde{\mathbf{x}}, \mathbf{m})$.

4.2.2 Batch Augmentation

Batch augmentation (Hoffer et al., 2019) duplicates a portion of training examples within the same batch with different perturbations. It differs slightly from data augmentation, where a batch contains p perturbed and $1 - p$ original data. In contrast, batch augmentation *augments* the entire batch with p perturbed rather than *replacing* p original data with perturbed data as in data augmentation. Given a training batch and its paired perturbed batch, the objective of batch augmentation is defined as follows,

$$\mathcal{L}_{\mathrm{MBA}} = \sum_{j=1}^{b} a_j \mathcal{L}_{\mathrm{CLM}}(\tilde{\mathbf{x}}^j, \tilde{\mathbf{m}}^j) + \mathcal{L}_{\mathrm{CLM}}(\mathbf{x}^j)$$

When batch augmentation was originally proposed, its goal was not to improve the model robustness. We hypothesize that batch augmentation can further improve the robustness over data augmentation, as indicated in some multilingual cases (Ahmed and Devanbu, 2022; Wang et al., 2023a).

4.2.3 Adversarial Logits Pairing

Adversarial Logits Pairing (ALP) (Kannan et al., 2018) improves the robustness of classification models by minimizing the KL divergence between the original input's prediction distribution and the perturbed input's prediction distribution. However, adapting ALP from classification models to generation models trained by CLM is challenging. One straightforward approach is decomposing the generation task into multiple next-token prediction tasks. However, the original and the perturbed token sequences can have different lengths due to some transformations adding or removing tokens. This length discrepancy creates mismatches of each token's prediction between the two sequences.

To address this challenge, we leverage the sequence-level pairing provided by our paired dataset generation method. We apply ALP only to the unperturbed segments of the two sequences, marked by U in Example 4.2. All unperturbed segments have the same length, allowing us to apply ALP to the predictions of these unperturbed tokens. We use $\mathbf{u}$ and $\tilde{\mathbf{u}}$ to denote the ordered indices for all unperturbed tokens in the original and perturbed sequences. The ALP objective is defined as follows,

$$\mathcal{L}_{\text{ALP}} = \sum_{j=1}^{b} \sum_{i=1}^{|\mathbf{u}^j|} D_{\text{KL}} \left(p_f(\cdot \mid \tilde{\mathbf{x}}^j_{:\tilde{u}^j_i}) \parallel p_f(\cdot \mid \mathbf{x}^j_{:u^j_i}) \right)$$

Example 4.3. *In Example 4.1,* $\mathbf{u} = (0, 2, 3, 4)$ *and* $\tilde{\mathbf{u}} = (0, 3, 4, 5)$.

ALP with name-Dropout (ALPD) We design another ALP objective (ALPD) specifically tailored to variable and function renaming among context-sensitive perturbations. ALPD reduces the model's reliance on specific variable and function names by setting the attention masks of a portion of these names to zero. It can be seen as a dropout mechanism specific to entity names.

We use $\mathrm{Dp}(\mathbf{x})$ to denote the input sequence after name-specific dropout.

$$\begin{aligned}\mathcal{L}_{\text{ALPD}} = \sum_{j=1}^{b} \sum_{i=1}^{|\mathbf{u}^j|} & D_{\text{KL}} \left(p_f(\cdot \mid \mathrm{Dp}(\mathbf{x}^j)_{:u^j_i}) \parallel p_f(\cdot \mid \mathbf{x}^j_{:u^j_i}) \right) \\ & + D_{\text{KL}} \left(p_f(\cdot \mid \mathrm{Dp}(\tilde{\mathbf{x}}^j)_{:\tilde{u}^j_i}) \parallel p_f(\cdot \mid \tilde{\mathbf{x}}^j_{:\tilde{u}^j_i}) \right)\end{aligned}$$

$\mathcal{L}_{\text{ALPD}}$ sums two KL divergence losses: the first over the original sequence after and before dropout, and the second over the perturbed sequence.

4.2.4 Contrastive Learning

Contrastive learning (CL) maximizes the cosine similarity between positive (similar) pairs and minimizes the distance between negative (dissimilar) pairs. The granularity of pairs leads to different designs of CL objectives. This section introduces three designs of CL objectives tailored to CLM. ContraSeq and ContraToken, inspired by ContraCLM (Jain et al., 2023), focus on sequences and tokens, respectively. A novel ContraName objective focuses on variable and function names.

ContraSeq The ContraSeq objective operates at the sequence level, where each pair consists of summarizations of two input sequences. We note that ContraSeq is also adopted in ContraBERT (Liu et al., 2023) and ContraCode (Jain et al., 2021) for improving the robustness of the encoder model trained on masked language modeling (MLM). Since CLM does not have the [CLS] token used in MLM, we compute the average of the hidden states in the last layer as the summarization.

Given a batch $B = \{\mathbf{h}_1, \ldots, \mathbf{h}_b, \tilde{\mathbf{h}}_1, \ldots, \tilde{\mathbf{h}}_b\}$ with $2b$ summarizations of original and perturbed sequences, ContraSeq treats the b corresponding original and perturb pairs, i.e., $(\mathbf{h}_i, \tilde{\mathbf{h}}_i)$, as positive pairs and other pairs in the batch as negatives.

Denoting the temperature hyper-parameter as τ and cosine similarity as $\diamond$, we define the ContraSeq objective as follows,

$$\mathcal{L}_{\text{CSeq}} = \sum_{j=1}^{b} g(\mathbf{h}^j, \tilde{\mathbf{h}}^j, B) + g(\tilde{\mathbf{h}}^j, \mathbf{h}^j, B),$$

where $g(x, y, B)$ is defined as

$$g(x, y, B) = -\log \frac{\exp(x \diamond y / \tau)}{\sum_{h \in B} \exp(x \diamond h / \tau) - \exp(1/\tau)}.$$

ContraSeq represents the coarsest granularity among the three objectives. While ContraSeq is shown to be effective for MLM, it may not fully cater to CLM, which predicts the next token for each prefix and involves discriminating representations at finer levels, e.g., tokens and names. Additionally, ContraSeq poses scalability challenges, as it demands a large batch size to compute a meaningful InfoNCE loss. This challenge restricts ContraSeq's feasibility to large language models.

ContraToken The ContraToken objective operates at the token level, providing a finer granularity than ContraSeq. ContraToken aims to discriminate the representation of each prefix. However, as we mentioned in Section 4.2.3, directly treating $\mathbf{x}_{:i}$ and $\tilde{\mathbf{x}}_{:i}$ as a positive pair does not work due to the potential sequence length difference between original and perturbed sequences. To address this, ContraToken considers two prefixes ending at the same unperturbed token as a positive pair, with other prefix pairs designated as negatives.

For the j-th training example, let $\mathbf{h}_i^j$ denote the representation of the prefix up to the i-th token, and u_i^j denote the index of the i-th unperturbed token. We define ContraToken objective as:

$$\mathcal{L}_{\text{CTok}} = \sum_{j=1}^{b} \sum_{i=1}^{|\mathbf{u}^j|} g(\mathbf{h}_{u_i^j}^j, \tilde{\mathbf{h}}_{\tilde{u}_i^j}^j, H^j) + g(\tilde{\mathbf{h}}_{\tilde{u}_i^j}^j, \mathbf{h}_{u_i^j}^j, H^j)$$

where H^j contains all the representations of prefixes ending at unperturbed tokens for the j-th training example, i.e., $H^j = \{\mathbf{h}_{u_1^j}^j, \ldots, \mathbf{h}_{u_{|\mathbf{u}^j|}^j}^j, \tilde{\mathbf{h}}_{\tilde{u}_1^j}^j, \ldots, \tilde{\mathbf{h}}_{\tilde{u}_{|\tilde{\mathbf{u}}^j|}^j}^j\}$.

ContraName We design a novel name-level CL objective, ContraName, to address variable and function renaming in context-sensitive perturbations. It aims to enhance the discrimination of representations of variable and function names.

In ContraName, we group representations of variables or functions according to their names. For a name spanning multiple tokens, we use the average of these tokens as its representation. ContraName treats representations within the same group as positive pairs and those across different groups as negative pairs. Notice that the negative pairs in ContraName have explicit semantic differences, i.e., different names should yield different representations. This explicit semantic difference of negative pairs has been shown to improve the effectiveness of CL (Ding et al., 2023).

Suppose in the sequence $\mathbf{x}$, we identify g groups of name representations $G_1, G_2, \ldots, G_g$, with $G = \bigcup_{i=1}^{g} G_i$ being their union. We define the ContraName objective on the input $\mathbf{x}$ as follows,

$$\mathcal{L}_{\text{CName}}(\mathbf{x}) = -\log \left(\frac{\sum_{i=1}^{g} \sum_{\mathbf{h},\mathbf{h}' \in G_i} \exp(\mathbf{h} \diamond \mathbf{h}'/\tau)}{\sum_{\mathbf{h},\mathbf{h}' \in G} \exp(\mathbf{h} \diamond \mathbf{h}'/\tau)} \right)$$

Example 4.4. *Consider the original example in Example 4.1 with* $G_1 = \{\mathbf{h}_1, \mathbf{h}_7\}$ *and* $G_2 = \{\mathbf{h}_8\}$. *The perturbed example contains* $\tilde{G}_1 = \{\tilde{\mathbf{h}}_2, \tilde{\mathbf{h}}_7\}$ *and* $\tilde{G}_2 = \{\frac{\tilde{\mathbf{h}}_8 + \tilde{\mathbf{h}}_9}{2}\}$.

The final ContraName objective is the sum of losses over the original sequences and their perturbed counterparts.

$$\mathcal{L}_{\text{CName}} = \sum_{j=1}^{m} \mathcal{L}_{\text{CName}}(\mathbf{x}^j) + \mathcal{L}_{\text{CName}}(\tilde{\mathbf{x}}^j) \quad (6)$$

5 Evaluation

5.1 General Experimental Setup

Models We use different robust training approaches to fine-tune different sizes of monolingual CodeGen models (Nijkamp et al., 2023b) targeting at Python: CodeGen-6B, 2B, and 350M. We provide fine-tuning settings in Appendix A.

Datasets and Benchmarks We use the Stack dataset v1.2 (Kocetkov et al., 2022) as our original training dataset. We uses ReCode (Wang et al., 2023b) to augment the dataset by introducing different code perturbations. For all experiments, we set $p = 25\%$ and $t = 2$ (Eq 4), i.e., we apply at most two perturbations to each original code snippet.

To evaluate the model robustness, we use the ReCode benchmark, which is based on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) with a total of 1138 (164 + 974) problems. The docstring and function-name classes are perturbed based on the original prompt. The code-syntax and code-format classes are perturbed based on modified prompts, which are appended with half of the ground truth completion.

Metrics We use the following three metrics to assess the model performance.

NP@1. We use Pass@k following Chen et al. (2021) to assess the nominal code generation performance. We name the Pass@k on unperturbed data as Nominal Pass@k (NP@k). This metric approximates the probability of any k samples passes all the test case, if we randomly choose k samples out of n samples generated by the model for each problem. We use $n = 5$ because the difference between $n = 10$ and $n = 100$ is already small as demonstrated in Recode (Wang et al., 2023b).

RP$_{10}$@1. To evaluate the robustness of models, we use the Robust Pass$_s$@k (RP$_s$@k). It measures the worst-case Pass@k on s perturbed variance for each perturbation type and each sample. Here, we use $s = 10$ to harden the robustness gain for training and differentiate performance gaps.

Drop%. We report Robust Drop%. It measures the percentage drop from Robustness Pass@k (RP$_s$@k) from Nominal Pass@k (NP@k), indicating the relative robustness changes given perturbations. Lower Drop% means better robustness.

Model & Methods		Docstring		Function		Syntax		Format		Overall Average		
		NP@1	RP$_{10}$@1	NP@1	RP$_{10}$@1	NP@1	RP$_{10}$@1	NP@1	RP$_{10}$@1	NP@1	RP$_{10}$@1	Drop%
CodeGen-6B	Ori	35.96	12.83	35.96	14.36	52.72	2.20	52.72	25.47	44.34	13.71	69.08
	$\mathcal{L}_{\text{CLM}}$	40.07	20.21	40.07	22.18	54.91	2.58	54.91	35.80	**47.27**	20.19	57.29
	$\mathcal{L}_{\text{DA}}$	37.61	20.51	37.61	21.88	52.99	27.66	52.99	42.18	45.30	28.06	38.06
	Ours	37.91	**23.13	37.91	**22.53**	53.16	**27.70**	53.16	**44.87	45.54	**29.56**	**35.09**
CodeGen-2B	Ori	31.27	11.04	31.27	9.75	44.82	1.63	44.82	24.45	38.05	11.72	69.20
	$\mathcal{L}_{\text{CLM}}$	32.99	15.78	32.99	16.41	46.22	2.43	46.22	32.00	**39.61**	16.66	57.94
	$\mathcal{L}_{\text{DA}}$	31.62	17.62	31.62	16.61	45.96	*22.86	45.96	**34.50**	38.79	22.90	40.96
	Ours	31.56	**19.21	31.56	**17.12**	45.04	21.92	45.04	34.36	38.30	**23.15**	**39.56**
CodeGen-350M	Ori	17.10	3.57	17.10	3.06	26.75	1.11	26.75	9.54	21.93	4.32	80.30
	$\mathcal{L}_{\text{CLM}}$	18.10	6.19	18.10	6.47	29.24	1.46	29.24	15.96	23.67	7.52	68.23
	$\mathcal{L}_{\text{DA}}$	18.10	7.45	18.10	7.94	30.11	8.59	30.11	18.58	24.10	10.64	55.85
	Ours	18.33	**9.72	18.33	**8.05**	31.04	**10.67	31.04	**21.58	**24.69**	**12.51**	**49.33**

Table 1: Robust evaluation of CodeGen-6B, 2B, 350M on the ReCode benchmark. Our approach combines $\mathcal{L}_{\text{MBA}}$, $\mathcal{L}_{\text{ALP}}$, and $\mathcal{L}_{\text{ALPD}}$. We show the statistical significance between our approach and $\mathcal{L}_{\text{DA}}$ using the paired-t test with * denoting $p < 0.05$ and ** denoting $p < 0.01$. NP@1 and RP$_{10}$@1 are higher the better. Drop% is lower the better.

5.2 Effectiveness of Proposed Approaches

Summary of Results: Our approach significantly enhances the robustness of code generation models, surpassing the results of data augmentation. Notably, our approach exhibits the most substantial improvement in robustness against Syntax perturbations.

Table 1 summarizes the robust evaluation results for CodeGen models. We use $\mathcal{L}_{\text{CLM}}$ to denote the baseline method fine-tuning on the original stack dataset (unseen by CodeGen models) without any robust training approaches. Comparing $\mathcal{L}_{\text{CLM}}$ and the original model (Ori), we find that fine-tuning on unseen data can already improve model robustness on the Docstring, Function, and Format perturbations, except the Syntax perturbation.

When averaging across four perturbation classes, our approach demonstrates significant improvements in RP$_{10}$@1 —9.37, 6.49, and 4.99 for CodeGen-6B, CodeGen-2B, and CodeGen-350M, respectively, compared to the baseline $\mathcal{L}_{\text{CLM}}$. In contrast, data augmentation achieves sub-optimal results with improvements of 7.87, 6.24, and 3.12.

Averaging over all three models, our approach enhances RP$_{10}$@1 by 3.29, 0.88, 17.94, and 5.68 for Docstring, Function, Syntax, and Format perturbations, respectively, compared to the baseline $\mathcal{L}_{\text{CLM}}$. Surprisingly, our approach exhibits the most substantial improvement in robustness against Syntax perturbations. This emphasis on strengthening robustness to Syntax perturbations is crucial for ensuring the reliability of code models in handling diverse syntactic variations.

We conducted statistical analyses using paired-t tests to compare our approach with baseline $\mathcal{L}_{\text{CLM}}$ and data augmentation $\mathcal{L}_{\text{DA}}$ across four perturbation classes. Our approach significantly outperforms the baseline $\mathcal{L}_{\text{CLM}}$ with $p < 0.05$ on all perturbation classes and all models with exceptions of function-name perturbations on CodeGen-6B and CodeGen-2B. We hypothesize that the less pronounced results on function-name perturbations are due to the imbalanced perturbed data, as the percentages of function-name perturbations are much smaller compared to other perturbation types. When comparing our approach with data augmentation $\mathcal{L}_{\text{DA}}$ (shown in Table 1), we found that our approach significantly outperforms $\mathcal{L}_{\text{DA}}$ with $p < 0.01$ on six cases, while $\mathcal{L}_{\text{DA}}$ outperforms our approach with $p < 0.05$ on one case.

5.3 Ablation Studies

Summary of Results: The ablation studies confirm the effectiveness of masked batch augmentation, ALP, and ALPD. ContraSeq provides negligible improvements compared to the baseline ($\mathcal{L}_{\text{CLM}}$). ContraToken and ContraName yield mixed results in different settings.

This section presents the ablation results of different approaches outlined in Section 4.2 applied to the CodeGen-350M model. We conduct our experiments in two settings. The first setting (Table 2) focuses on context-free perturbations, applying the original data augmentation ($\mathcal{L}_{\text{DA}}$) loss to context-sensitive perturbations while varying different approaches for the context-free perturbations. In the second setting (Table 3), we vary approaches for context-sensitive perturbations while maintaining the $\mathcal{L}_{\text{DA}}$ loss for context-free perturbations. We report the overall average of NP@1, RP$_{10}$@1, and Drop% across four perturbation classes, with more

Methods	Overall Average		
	NP@1	RP_{10}@1	Drop%
[0] : $\mathcal{L}_{\text{CLM}}$	23.67	7.52	68.23
[1] : $\mathcal{L}_{\text{DA}}$	24.10	10.64	55.85
[2] : $\mathcal{L}_{\text{MDA}}$	23.77	11.10	53.30
[3] : $\mathcal{L}_{\text{MDA}}(p = 20\%)$	24.60	10.48	57.40
[4] : $\mathcal{L}_{\text{MBA}}$	24.90	11.01	55.78
[5] : $\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALP}}$	24.67	11.48	53.47
[6] : $\mathcal{L}_{\text{CLM}} + \mathcal{L}_{\text{CSeq}}$	23.80	7.79	67.27
[7] : [5] + $\mathcal{L}_{\text{CTok}}$	23.52	11.18	52.47
[8] : [5] + $\mathcal{L}_{\text{CSeq}}$	24.64	11.51	53.29

Table 2: Ablation results on *context-free perturbations*.

Methods	Overall Average		
	NP@1	RP_{10}@1	Drop%
[0] : $\mathcal{L}_{\text{CLM}}$	23.67	7.52	68.23
[1] : $\mathcal{L}_{\text{MBA}}$	24.83	10.75	56.71
[2] : $\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALPD}}$	24.82	10.94	55.92
[3] : $\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{CName}}$	24.60	10.87	55.81
[4] : [2] + $\mathcal{L}_{\text{CName}}$	24.66	10.80	56.20
[5] : [4] + $\mathcal{L}_{\text{CTok}}$	24.42	10.42	57.33

Table 3: Ablation results on *context-sensitive* perturbations.

details reported in Appendix E.

Effectiveness on Masked Batch Augmentation The masked batch augmentation loss $\mathcal{L}_{\text{MBA}}$ consists of two components: (1) a masking mechanism that masks unnatural perturbed tokens and (2) batch augmentation. Comparing the results of masked data augmentation ($\mathcal{L}_{\text{MDA}}$) and data augmentation ($\mathcal{L}_{\text{DA}}$) in Table 2 validates the effectiveness of the masking mechanism because $\mathcal{L}_{\text{MDA}}$ achieves better RP_{10}@1 and Drop% than $\mathcal{L}_{\text{DA}}$. To assess the effectiveness of batch augmentation, we cannot directly compare the results of $\mathcal{L}_{\text{MDA}}$ ([2], Table 2) and $\mathcal{L}_{\text{MBA}}$ ([4], Table 2) because $\mathcal{L}_{\text{MDA}}$ is trained on $p = 25\%$ perturbed data, while $\mathcal{L}_{\text{MBA}}$ is trained on $\frac{p}{1+p} = 20\%$ perturbed data. For a fair comparison, we train $\mathcal{L}_{\text{DA}}$ with $p = 20\%$ perturbed data and report the result at [3] in Table 2. Comparing the results of $\mathcal{L}_{\text{MDA}}(p = 20\%)$ and $\mathcal{L}_{\text{MBA}}$ confirms the effectiveness of batch augmentation because $\mathcal{L}_{\text{MBA}}$ achieves better NP@1, RP_{10}@1, and Drop% than $\mathcal{L}_{\text{MDA}}(p = 20\%)$.

Effectiveness of ALP and ALPD ALP and ALPD are shown to be effective because $\mathcal{L}_{\text{ALP}}$ and $\mathcal{L}_{\text{ALPD}}$ both improve the RP_{10}@1 and Drop% ([5] vs [4] in Table 2 and [1] vs [2] in Table 3). We further investigate different designs of ALP and ALPD in Appendix E.

Discussion on Contrastive Learning Objectives ContraSeq only provides negligible improvements, as evidenced by [6] vs [0], and [8] vs [5] in Table 2. ContraToken behaves differently in two ablation experiment settings. In the context-free perturbation experiment (Table 2), ContraToken improves the Drop% but negatively impacts the NP@1 and RP_{10}@1 ([7] vs [5]). Conversely, ContraToken hurts all metrics ([5] vs [4]) in the context-sensitive perturbation experiment (Table 3). Adding ContraName to the masked batch augmentation loss improves the Drop% and RP_{10}@1 ([3] vs [1]).

6 Conclusion

We propose CodeFort, a framework to improve the robustness of code generation models, generalizing a large variety of code perturbations to enrich the training data and enabling various robust training strategies. Our approach significantly enhances the model robustness and surpasses the sub-optimal results of data augmentation. Notably, our approach significantly improves the robustness under code-syntax perturbations, the type of perturbation that hurts the model robustness the most. Our ablation studies show that ContraSeq, the CL objective used in previous work for MLM, has negligible robustness improvements on CLM.

7 Limitations and Future Work

We foresee many future improvements to this paper. First, ALPD and ContraName primarily target function and variable rename perturbations but are not general enough to handle arbitrary context-sensitive perturbations. However, these approaches can be applied to name-entities in general NLP tasks. Second, the robustness improvement of function-name perturbation on CodeGen-6B and CodeGen-2B is insignificant compared to the baseline, necessitating unique strategies to overcome this limitation. Thirdly, our evaluation is limited to the CodeGen model architecture and primarily uses popular benchmarks like HumanEval and MBPP. However, we have assessed our approach across three different sizes of CodeGen models to illustrate its generalizability. Furthermore, it is important to note that our perturbed training dataset is generated based on real-world programs from the Stack v1.2 dataset. By training our models on a dataset that follows a real-world program distribution, we hypothesize that models trained using our approach can generalize effectively to other real-world coding benchmarks.

References

Toufique Ahmed and Premkumar T. Devanbu. 2022. Multilingual training for software engineering. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1443–1455. ACM.

Mrinal Anand, Pratik Kayal, and Mayank Singh. 2021. On adversarial robustness of synthetic code generation. *CoRR*, abs/2106.11629.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Pavol Bielik and Martin T. Vechev. 2020. Adversarial robustness for code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 896–907. PMLR.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Yangruibo Ding, Saikat Chakraborty, Luca Buratti, Saurabh Pujar, Alessandro Morari, Gail E. Kaiser, and Baishakhi Ray. 2023. CONCORD: clone-aware contrastive learning for source code. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 26–38. ACM.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Fengjuan Gao, Yu Wang, and Ke Wang. 2023. Discrete adversarial attack to models of code. *Proc. ACM Program. Lang.*, 7(PLDI).

Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas W. Reps. 2022. Semantic robustness of models of source code. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, pages 526–537. IEEE.

Elad Hoffer, Tal Ben-Nun, Itay Hubara, Niv Giladi, Torsten Hoefler, and Daniel Soudry. 2019. Augment your batch: better training with larger batches. *CoRR*, abs/1901.09335.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Nihal Jain, Dejiao Zhang, Wasi Uddin Ahmad, Zijian Wang, Feng Nan, Xiaopeng Li, Ming Tan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Xiaofei Ma, and Bing Xiang. 2023. Contraclm: Contrastive learning for causal language model. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 6436–6459. Association for Computational Linguistics.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 5954–5971. Association for Computational Linguistics.

Akshita Jha and Chandan K. Reddy. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 14892–14900. AAAI Press.

Harini Kannan, Alexey Kurakin, and Ian J. Goodfellow. 2018. Adversarial logit pairing. *CoRR*, abs/1803.06373.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code. *Preprint*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2476–2487. IEEE.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

George A. Miller. 1992. WordNet: A lexical database for English. In *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages. *ICLR*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating adversarial computer programs using optimized obfuscations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Sahil Suneja, Yufan Zhuang, Yunhui Zheng, Jim Laredo, Alessandro Morari, and Udayan Khurana. 2023. Incorporating signal awareness in source code modeling: An application to vulnerability detection. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Zhao Tian, Junjie Chen, and Zhi Jin. 2023. Adversarial attacks on neural models of code via code difference reduction. *CoRR*, abs/2301.02412.

Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023a. One adapter for all programming languages? adapter tuning for code search and summarization. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 5–16. IEEE.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023b. Recode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13818–13843. Association for Computational Linguistics.

Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. 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*, pages 1482–1493. ACM.

Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.*, 4(OOPSLA):162:1–162:30.

Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi

Jin. 2022. Towards robustness of deep program processing models - detection, estimation, and enhancement. *ACM Trans. Softw. Eng. Methodol.*, 31(3):50:1–50:40.

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1169–1176. AAAI Press.

Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2022. Adversarial robustness of deep code comment generation. *ACM Trans. Softw. Eng. Methodol.*, 31(4):60:1–60:30.

Transformation		StarCoder	WizardCoder	CodeGen16B
Docstring	NP@1	41.27	53.29	39.23
	RP_5@1	11.60	20.43	15.81
	Drop%	71.89	61.66	69.70
Function	NP@1	41.27	53.29	39.23
	RP_5@1	15.30	29.06	26.95
	Robust%	62.93	45.47	31.30
Syntax	NP@1	59.34	61.09	56.78
	RP_5@1	4.21	9.86	5.54
	Robust%	92.91	83.86	90.24
Format	NP@1	59.34	61.09	56.78
	RP_5@1	23.61	28.13	39.59
	Robust%	60.21	53.95	30.27

Table 4: The ReCode (Wang et al., 2023b) robustness evaluation for SOTA public code models. NP@1 shows the nominal pass@1 without perturbation; RP_5@1 shows the robust pass@1 under perturbation. The significant drop of Drop% indicates unsatisfied robustness performance of these models.

A Experiment Setups

Experiment Environment All fine-tuning experiments run on a cluster of Amazon EC2 P4 Instances. All evaluation experiments on ReCode run on a cluster of Amazon EC2 P4 Instances and Amazon EC2 P3 Instances.

Fine-tuning Settings We train with $p = 25\%$ perturbed data as CodeGen models has not been fine-tuned on the stack dataset. For CodeGen-2B and CodeGen-6B, we set batch size to 256 and fine-tune them for 10K and 5K steps, respectively, using the AdamW optimizer and a linear schedule with 500 warmup steps and a learning rate 2×10^{-5}. For CodeGen-350M, we set batch size to 512 and fine-tune the model on half of the stack dataset (about 266K steps) using the FusedAdam optimizer and a linear schedule with 500 warmup steps and a learning rate 2×10^{-5}.

We treat all the objective functions proposed in this paper equally, i.e., summing them up without reweighing. For the temperature hyperparameter τ in contrastive learning, we set $\tau = 0.05$ for all experiments following ContraCLM. We set the dropout rate to 0.1 for $\mathcal{L}_{\text{ALPD}}$.

Training Cost of Proposed Approaches We apply our approach to a subset of the training data, specifically 25% of the examples. For this subset, our approach requires twice as much memory as standard data augmentation because it needs to see both the perturbed and the original examples simultaneously. The rest of the training costs are the same as data augmentation.

It is important to note that for the remaining

$1 - 25\% = 75\%$ of the training data, our approach has the same training cost as standard data augmentation. Considering the benefits of improved model robustness, the overall increase in training cost is relatively modest. Further, users can trade off the training cost and targeted robustness gain by adjusting p.

Discussion of the Licenses of Datasets In our paper, we employed 1) the HumanEval dataset which is distributed under the MIT license, 2) the MBPP dataset, which is under the Apache-2.0 license, 3) the CodeGen model, which is governed by the BSD-3-Clause license, and 4) the stack v1.2 dataset comprised of a collection of permissively-licensed source code.

B Discussion on Adversarial Attacks and Adversarial Training

Numerous adversarial attacks have targeted encoder-decoder models in code-related tasks, including classification (e.g., vulnerability prediction) and generation (e.g., code summarization). Key methods include CODA (Tian et al., 2023), which exploits syntactic differences for adversarial example generation; CARROT (Zhang et al., 2022), employing a lightweight hill climbing for optimization in attacks; and ALERT (Yang et al., 2022), which creates naturalness-aware attacks using pre-trained models.

Existing work typically enhances model robustness through data augmentation and adversarial training (Madry et al., 2018). Bielik and Vechev (2020) refine model representations by feeding only pertinent program parts to the model; Suneja et al. (2023) use curriculum learning and data augmentation with simplified programs.

Our experiments did not include adversarial training due to its significant computational overhead. For example, consider CodeAttack (Jha and Reddy, 2023), an adversarial attack method for pre-trained code models. Given an input with length n and each token having l possible substitutions, CodeAttack requires $O(nl)$ forward evaluations of the model in the worst-case scenario ($O(n)$ in the best-case) for each input to generate an adversarial example. Code generation models typically use a context-window size of $n = 4096$, which means that adversarial training using CodeAttack would introduce a training overhead of at least 4096 forward evaluations for each input. This overhead is prohibitively expensive for fine-tuning large language models with billions of parameters.

C Detailed Results for Each Perturbation Type

Table 5 shows a detailed breakdown of robustness gain by finetuning with our approach for each perturbation type evaluated on 350M, 2B, and 6B CodeGen models.

D Qualitative Examples

In this section, we present qualitative examples to demonstrate the robustness improvements of our robust trained models. On these MBPP examples, 6B CodeGen baseline model fails to generate correct completions after applying the perturbations. Our robust trained model, on the other hand, can still successfully complete these problems. Here, we list examples for the top four perturbation types that we have achieved the most improvements (detailed numbers for each perturbation type can be found in Table 5).

DeadCode Insertion. For this perturbation type, ReCode perturbation will insert a redundant code block including an `if condition`, a `for loop`, or a `while loop`. The models will usually be distracted by the inserted blocks, causing failure of completions. Here, in Listing 1, ReCode perturbation inserts a redundant `if condition` block at line 12-13. Even though the semantic meaning barely changes after this insertion, the 6B baseline model fails to generate the correct answer.

```
1  def test_distinct(data):
2      """
3      Write a python function to determine whether all the numbers are different from each other are not.
4      >>> test_distinct([1,5,7,9])
5      True
6      >>> test_distinct([2,4,5,5,7,9])
7      False
8      >>> test_distinct([1,2,3])
9      True
10     """
11     if len(data) == len(set(data)):
12         if False:
13             return True
14 
15 # === completion by baseline model ===
16     return False
17 
18 # === completion by our model ===
19         return True
20     else:
21         return False
```

Listing 1: DeadCoderInsertion (6B CodeGen)

Categories	Transformations	CodeGen 350M			CodeGen 2B			CodeGen 6B		
		$\mathcal{L}_{\mathrm{CLM}}$	$\mathcal{L}_{\mathrm{DA}}$	Ours	$\mathcal{L}_{\mathrm{CLM}}$	$\mathcal{L}_{\mathrm{DA}}$	Ours	$\mathcal{L}_{\mathrm{CLM}}$	$\mathcal{L}_{\mathrm{DA}}$	Ours
Nominal	Regular	18.10	18.10	18.33	32.99	31.62	31.56	40.07	37.61	37.91
	Partial	29.24	30.11	31.04	46.22	45.96	45.04	54.46	52.99	53.16
Docstring	BackTranslation	17.35	17.66	17.79	31.6	29.86	30.53	38.91	36.75	37.45
	EnglishInflectionalVariation	10.98	10.98	12.50	23.95	22.93	23.99	28.35	27.21	29.02
	SynonymSubstitution	7.03	8.98	11.20	17.35	18.95	21.18	22.11	22.78	25.06
	TenseTransformationFuture	17.49	17.72	18.33	32.07	31.34	31.35	39.79	37.38	37.63
	TenseTransformationPast	18.12	18.51	18.63	32.55	31.21	31.55	39.40	37.28	37.70
	WorstCase	6.19	7.45	9.72	15.78	17.07	19.21	20.21	20.51	23.13
Function	RenameButterFinger	11.56	11.79	11.90	23.88	23.15	23.34	29.86	28.95	28.82
	RenameCamelCase	17.70	17.72	18.15	34.08	32.06	32.37	40.47	37.91	38.59
	RenameChangeChar	8.54	10.39	10.33	20.14	20.91	20.88	27.03	26.63	26.84
	RenameInflectionalVariation	14.11	14.76	15.20	28.56	27.87	28.19	33.36	32.48	33.66
	RenameSwapChar	11.92	11.86	12.20	24.69	24.17	24.25	31.44	29.81	29.49
	RenameSynonymSub	12.07	12.95	13.04	24.97	24.50	24.82	30.14	29.95	30.47
	WorstCase	6.47	7.94	8.05	16.41	16.61	17.12	22.18	21.88	22.53
Syntax	DeadCodeInsertion	1.92	15.83	20.77	3.87	33.25	32.93	3.32	38.86	41.16
	DeadCodeInsertionLast	9.24	31.55	32.69	13.90	48.26	49.47	14.39	55.13	55.15
	ForWhileTransformer	27.08	26.99	29.16	43.78	42.76	41.90	50.35	50.49	50.81
	OperandSwap	27.80	26.91	29.12	44.50	43.32	43.15	51.53	51.46	51.60
	VarRenamerCB	26.52	25.85	27.72	44.60	42.85	42.04	49.12	48.17	49.35
	VarRenamerNaive	24.99	26.31	26.22	42.53	41.20	41.09	49.28	49.05	48.14
	VarRenamerRN	14.75	15.41	15.78	31.65	31.48	30.56	37.07	37.07	36.03
	WorstCase	1.46	8.59	10.67	2.43	22.86	21.92	2.58	27.66	27.70
Format	Doc2Comments	25.48	27.66	28.75	45.36	44.29	42.67	50.28	51.27	51.30
	NewLineInsertion	20.44	22.32	25.54	35.52	37.86	37.91	39.74	46.19	48.58
	SplitLine	27.07	28.42	30.04	44.60	45.18	43.44	52.07	52.16	51.83
	WorstCase	15.96	18.58	21.58	32.00	34.50	34.36	35.80	42.18	44.87

Table 5: Robustness evaluation for each category of perturbations on combined HumanEval and MBPP datasets. We highlight in gray the top four perturbation types that we have achieved the most improvements over the baseline $\mathcal{L}_{\mathrm{CLM}}$.

NewLine Insertion. For newline insertions, ReCode will randomly insert empty lines within prompts, and code models commonly perform poor robustness on newline insertions as reported in (Wang et al., 2023b). In Listing 2, 6B baseline model will generate low quality outputs if we simply insert two single empty lines at line 12 and line 15. Our models remain correct under the perturbation.

```
1  def count_first_elements(test_tup):
2    """
3    Write a function to find the element count that occurs before the record in the given tuple.
4    >>> count_first_elements((1, 5, 7, (4, 6), 10) )
5    3
6    >>> count_first_elements((2, 9, (5, 7), 11) )
7    2
8    >>> count_first_elements((11, 15, 5, 8, (2, 3), 8) )
9    4
10   """
11
12
13   for count, ele in enumerate(test_tup):
14
15
16 # === completion by baseline model ===
17     if isinstance(ele, tuple):
18       return count_first_elements(ele)
19
20     else:
21       return count
22
23 # === completion by our model ===
24     if isinstance(ele, tuple):
25       return count
26   return len(test_tup)
```

Listing 2: NewlineInsertion (6B CodeGen)

Synonym Substitution. To perform synonym substitution on docstrings, ReCode will randomly select words within docstrings or comments. It then replaces the selected words to their synonyms according to WordNet (Miller, 1992). In Listing 3, “have" is replaced to be “induce" and “given" to “yield". With only these two-word changes, 6B baseline model fails. In contrast, our model is much more robust to these changes.

```
def find_tuples(test_list, K):
  """
  Write a function to find tuples which induce all element divisible by k from the yield list of tuples.
```

```
    >>> find_tuples([(6, 24, 12), (7, 9, 6), (12, 18, 21)], 6)
    '[(6, 24, 12)]'
    >>> find_tuples([(5, 25, 30), (4, 2, 3), (7, 8, 9)], 5)
    '[(5, 25, 30)]'
    >>> find_tuples([(7, 9, 16), (8, 16, 4), (19, 17, 18)], 4)
    '[(8, 16, 4)]'
    """

    # === completion by baseline model ===
    return '[' + ','.join(map(str, filter(lambda x: x[0] % K == 0, test_list))) + ']'

    # === completion by our model ===
    return str(list(filter(lambda x: all(x[i] % K == 0 for i in range(len(x))), test_list)))
```

Listing 3: SynonymSubstitution (6B CodeGen)

CharCase Change for Function Name. For function name perturbation, ReCode CharCase change is to randomly replace characters in function name to their upper cases. In Listing 4, “remove_dirty_chars"" are replaced with “remOve_dIrty_cHarS". With only four characters changed to their upper cases, 6B baseline model will fail under such perturbation. Our model demonstrates better robustness performance.

```
def remOve_dIrty_cHarS(string, second_string):
        """
        Write a function to remove characters from the first string which are present in the second string.
        >>> remOve_dIrty_cHarS("probasscurve", "pros")
        'bacuve'
        >>> remOve_dIrty_cHarS("digitalindia", "talent")
        'digiidi'
        >>> remOve_dIrty_cHarS("exoticmiles", "toxic")
        'emles'
        """

# === completion by baseline model ===
        return string.replace(second_string, "")

# === completion by our model ===
        for char in second_string:
                if char in string:
                        string = string.replace(char, "")
        return string
```

Listing 4: CharCase change for function name (6B CodeGen)

E Detailed Results for Ablations

Tables 6 and 7 shows detailed comparison among different approaches across four perturbation classes.

E.1 Different Designs of ALP

This section compares different designs of ALP. In terms of the KL divergence loss, two approaches are considered: (1) optimizing both original and perturbed token prefixes simultaneously, i.e., bringing their output distributions closer at the same time, denoted as *Bo* (both sides), and (2) optimizing only the perturbed token prefix, i.e., only bringing the output distribution of the perturbed token prefix closer to the original one, denoted as *On* (one side). Another aspect involves whether to optimize all prefixes or just the ones that are correctly predicted. The instance that optimizes all prefixes is named *Al* (all), while the one optimizing only correctly predicted prefixes is named *CO* (correct only). In summary, there are four different ALP designs (two by two). Lines [9]-[12] in Table 6 show that *On*+*Al* achieves the best overall RP_{10}@1 among the four design. Therefore, we use this design throughout our experiments.

E.2 Different Designs of ALPD

This section compares three different designs of ALP. We conduct two additional experiments: (1) dropout of 10% arbitrary tokens, denoted as *All*, and (2) dropout of arbitrary tokens while following the same percentage as 10% of variable and function names, denoted as *AllS* (all stratified). Comparing line [2] with lines [6] and [7] in Table 7, we observe that $\mathcal{L}_{\mathrm{ALPD}}$ with 10% dropout on names achieves the best overall NP@1 and RP_{10}@1. Therefore, we use this design throughout our experiments.

E.3 Effectiveness of Combining Context-Free and Context-Sensitive Perturbations

Based on the ablation results, we choose to use $\mathcal{L}_{\mathrm{MBA}} + \mathcal{L}_{\mathrm{ALP}} + \mathcal{L}_{\mathrm{ALPD}}$ for all the models in Section 5.2. Our approach involves training on the combination of context-free and context-sensitive perturbations. Comparing the results of our combined approach on CodeGen-350M in Table 1 with those in Table 6 line [5] and in Table 7 line [2], we observe an improvement in model robustness. Specifically, our combined approach outperforms the other two approaches that focus solely on either

Methods	Docstring		Function		Syntax		Format		Overall Average		
	NP@1	RP_{10}@1	NP@1	RP_{10}@1	NP@1	RP_{10}@1	NP@1	RP_{10}@1	NP@1	RP_{10}@1	Drop%
[0]$\mathcal{L}_{\text{CLM}}$	18.10	6.19	18.10	6.47	29.24	1.46	29.24	15.96	23.67	7.52	68.23
[1]$\mathcal{L}_{\text{DA}}$	18.10	7.45	18.10	7.94	30.11	8.59	30.11	18.58	24.10	10.64	55.85
[2]$\mathcal{L}_{\text{MDA}}$	17.57	8.12	17.57	7.91	29.96	9.31	29.96	19.07	23.77	11.10	53.30
[3]$\mathcal{L}_{\text{MDA}}(p = 0.2)$	17.91	7.36	17.91	7.84	31.30	8.88	31.30	17.86	24.60	10.48	57.40
[4]$\mathcal{L}_{\text{MBA}}$	18.24	7.17	18.24	8.21	31.56	10.02	31.56	18.65	24.90	11.01	55.78
[5]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALP}}$	18.03	7.38	18.03	8.19	31.30	11.92	31.30	18.44	24.67	11.48	53.47
[6]$\mathcal{L}_{\text{CLM}} + \mathcal{L}_{\text{CSeq}}$	17.52	7.17	17.52	7.56	30.07	1.37	30.07	15.06	23.80	7.79	62.27
[7]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALP}} + \mathcal{L}_{\text{CTok}}$	17.33	7.12	17.33	8.03	29.72	11.46	29.72	18.10	23.52	11.18	52.47
[8]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALP}} + \mathcal{L}_{\text{CSeq}}$	17.98	7.38	17.98	8.15	31.30	11.81	31.30	18.70	24.64	11.51	53.29
[9]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALP}}(\textit{On} + \textit{Co}) + \mathcal{L}_{\text{CTok}} + \mathcal{L}_{\text{CSeq}}$	17.36	7.35	17.36	7.91	29.42	11.12	29.42	18.09	23.39	11.12	52.46
[10]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALP}}(\textit{On} + \textit{Al}) + \mathcal{L}_{\text{CTok}} + \mathcal{L}_{\text{CSeq}}$	17.31	7.33	17.31	7.80	29.77	11.48	29.77	18.12	23.54	11.18	52.51
[11]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALP}}(\textit{Bo} + \textit{Co}) + \mathcal{L}_{\text{CTok}} + \mathcal{L}_{\text{CSeq}}$	16.87	7.19	16.87	7.72	29.40	11.07	29.40	17.82	23.14	10.95	52.68
[12]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALP}}(\textit{Bo} + \textit{Al}) + \mathcal{L}_{\text{CTok}} + \mathcal{L}_{\text{CSeq}}$	16.70	7.15	16.70	7.77	29.54	11.41	29.54	17.62	23.12	10.99	52.47

Table 6: Ablation results of CodeGen-350M focusing on *context-free perturbations*, i.e., we apply $\mathcal{L}_{\text{DA}}$ loss to the context-sensitive perturbations except for the baseline $\mathcal{L}_{\text{CLM}}$.

Methods	Docstring		Function		Syntax		Format		Overall Average		
	NP@1	RP_{10}@1	NP@1	RP_{10}@1	NP@1	RP_{10}@1	NP@1	RP_{10}@1	NP@1	RP_{10}@1	Robust%
[0]$\mathcal{L}_{\text{CLM}}$	18.10	6.19	18.10	6.47	29.24	1.46	29.24	15.96	23.67	7.52	31.77
[1]$\mathcal{L}_{\text{MBA}}$	18.68	8.19	18.68	8.56	30.98	7.72	30.98	18.54	24.83	10.75	43.29
[2]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALPD}}$	18.80	8.44	18.80	8.37	30.84	8.00	30.84	18.98	24.82	10.94	44.08
[3]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{CName}}$	18.65	8.14	18.65	8.49	30.54	7.94	30.54	18.91	24.60	10.87	44.19
[4]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALPD}} + \mathcal{L}_{\text{CName}}$	18.63	8.12	18.63	8.26	30.69	7.82	30.69	18.98	24.66	10.80	43.80
[5]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALPD}} + \mathcal{L}_{\text{CName}} + \mathcal{L}_{\text{CTok}}$	18.31	7.79	18.31	7.80	30.53	8.17	30.53	17.93	24.42	10.42	42.67
[6]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALPD}}(\textit{All})$	18.07	8.66	18.07	8.12	30.33	7.56	30.33	18.88	24.20	10.80	44.63
[7]$\mathcal{L}_{\text{MBA}} + \mathcal{L}_{\text{ALPD}}(\textit{AllS})$	18.37	8.12	18.37	8.73	30.51	7.49	30.51	18.08	24.44	10.61	43.42

Table 7: Ablation results of CodeGen-350M focusing on *context-sensitive* perturbations, i.e., we apply $\mathcal{L}_{\text{DA}}$ loss to the context-free perturbations except for the baseline $\mathcal{L}_{\text{CLM}}$.

context-free or context-sensitive perturbations in Docstring and Format.