How Effectively Do Code Language Models Understand Poor-Readability Code?

Chao Hu
Shanghai Jiao Tong University
China
ythere@sjtu.edu.cn

Yitian Chai
Shanghai Jiao Tong University
China
sjtu_chaiyt@sjtu.edu.cn

Hao Zhou
WeChat AI, Tencent
China
tuxzhou@tencent.com

Fandong Meng
WeChat AI, Tencent
China
fandongmeng@tencent.com

Jie Zhou
Tencent Inc.
China
withtomzhou@tencent.com

Xiaodong Gu*
Shanghai Jiao Tong University
China
xiaodong.gu@sjtu.edu.cn

ABSTRACT

Code language models such as CodeT5 and CodeLlama have demonstrated substantial achievement in code comprehension. While the majority of research efforts have focused on improving model architectures and training processes, we find that the current benchmarks used for evaluating code comprehension models are confined to high-readability code, regardless of the popularity of low-readability code in reality. As such, they are inadequate to demonstrate the full spectrum of the model's ability, particularly the robustness to varying readability degrees. In this paper, we analyze the robustness of code summarization models to code with varying readability, including seven obfuscated datasets derived from existing benchmarks. Our findings indicate that current code summarization models are vulnerable to code with poor readability. In particular, their performance predominantly depends on semantic cues within the code, often neglecting the syntactic aspects. Existing benchmarks are biased toward evaluating semantic features, thereby overlooking the models' ability to understand nonsensitive syntactic features. Based on the findings, we present PoorCodeSumEval, a new evaluation benchmark on code summarization tasks. PoorCodeSumEval innovatively introduces readability into the testing process, considering semantic, syntactic, and their cross-obfuscation, thereby providing a more comprehensive and rigorous evaluation of code summarization models. Our studies also provide more insightful suggestions for future research, such as constructing multi-readability benchmarks to evaluate the robustness of models on poor-readability code, proposing readability-awareness metrics, and automatic methods for code data cleaning and normalization.

*Xiaodong Gu is the corresponding author.

KEYWORDS

Code language models, Code summarization, Code readability

ACM Reference Format:
Chao Hu, Yitian Chai, Hao Zhou, Fandong Meng, Jie Zhou, and Xiaodong Gu. 2024. How Effectively Do Code Language Models Understand Poor-Readability Code?. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695072

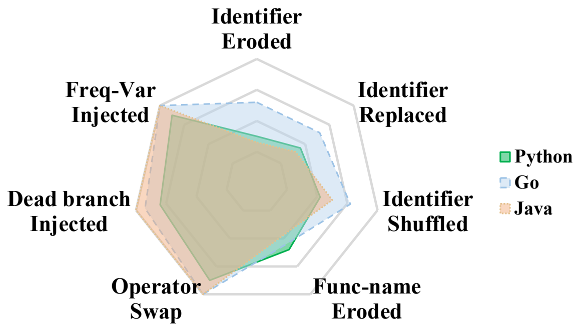


Figure 1: Fine-grained evaluation of CodeT5 across various readability in the code summarization task. The seven dimensions represent seven obfuscated datasets derived from existing benchmarks. The values presented in the figure correspond to the percentage of results obtained from each perturbed dataset relative to the results from the original benchmark. This figure allows us to quantify how changes in code readability affect the performance of CodeT5.

1 INTRODUCTION

Efficient program comprehension is crucial for developers and significantly enhances software development productivity. Code summarization, a process that generates natural language descriptions for source code, has witnessed substantial progress in recent years, primarily driven by the development of pre-trained language models such as CodeBERT, CodeT5, and CodeLlama [8, 26, 39].

While the majority of research efforts have focused on improving model architectures and training methodologies, it is apparent that the evaluation of code summarization models faces significant limitations. **First**, the test sets used for evaluating code summarization models are confined to high-readability code, characterized by well-structured syntax, meaningful variable names, and conventional coding styles. However, real-world software engineering often involves code with poor readability, as observed in various studies [1, 25, 31]. Unlike natural languages, source code exhibits varying formats and styles due to programmers adhering to different coding conventions and personal preferences. For instance, in reverse engineering, developers must comprehend decompiled code, which may lack meaningful original identifiers. Additionally, in the context of security, malicious code may be intentionally obfuscated by rearranging identifiers and structures to impede readability. To effectively address such challenges, it is crucial to assess the robustness of existing code summarization models against obfuscated code, which often features nonsense identifiers and dead code [1, 25, 31].

Second, and more critically, the current evaluation processes rely on single, holistic metrics, offering only a coarse-grained assessment of text similarity on a specific task. Consequently, these metrics fail to provide a comprehensive and intuitive demonstration of the fine-grained capabilities of code language models, particularly the model's robustness to varying readability. Figure 1 illustrates the robustness of CodeT5 in seven obfuscated test sets with different readability. The results indicate that current code language models are sensitive to various obfuscations.

In this paper, we provide a thorough evaluation of large language models in understanding low-readability code. We select a code benchmark with human-graded readability scores and partition it into low- and high-readability groups. We compare the performance of GPT-4o between the two groups. Our study indicates that readability has an evident impact on code comprehension by large language models. Particularly, the current LLMs show a strong vulnerability to poor readability code.

Having recognized this phenomenon, we further investigate what readability factors the current code LMs are most vulnerable to. We study finer-grained readability factors that target both semantics (such as identifiers and function names) and syntax (such as branches and symbols). For each factor, we gain different readability degrees through code obfuscation, i.e., perturbating code to reduce the readability while maintaining the semantics. We obfuscate the three primary benchmarks (TL-CodeSum [13], Deepcom [12], and CodeSearchNet [14]) into seven low-readability datasets according to widely used perturbation rules: Identifier Order Erosion (IOE), Identifier Random Shuffling (IRS), Identifier High-Frequency Replacement (IHR), Function Name Erosion (FNE), Operators and Operands Swap (OOS), Dead Branch Injection (DBI), and High-Frequency Variable Injection (HVI). Using the obfuscated datasets, we gain insights into the model's vulnerability to various perturbations. Our findings reveal that the current code summarization models rely heavily on the semantic cues of the code while struggling with comprehending structural features. Moreover, existing benchmarks are biased toward evaluating semantic features, thereby overlooking the models' ability to understand syntactic features.

Based on the findings, we propose a new cross-evaluation methodology to assess the comprehensive robustness of code summarization models. We create balanced-sensitivity datasets by combining two distinct sets of perturbation rules: one focusing on semantic perturbations and the other on syntactic perturbations. We select a semantic perturbation rule that has the maximal impact on the model performance and pairs it with one of the syntax perturbation rules. This allows the evaluation to minimize the impact of sensitive aspects while assessing the comprehensive ability of models on low-sensitive code features. We use this methodology to create a new benchmark named PoorCodeSumEval. Finally, we scrutinize the comprehensive ability of code summarization models using the results of both the robust evaluation and cross-evaluation of the balanced-sensitivity datasets.

The primary contributions of this paper are summarized as follows:

- We conduct an empirical study of the robustness of state-of-the-art code summarization models to poor readability code. Our results shed light on the comprehensive evaluation of code language models on varying readability features.
- We create an evaluation workflow to assess the comprehensive capability of code summarization models, taking into account their semantic, syntactic, and joint robustness. Our approach provides a comprehensive assessment of the robustness and adaptability of these models in real-world scenarios where code readability can significantly differ.
- We introduce a new benchmark dataset for a comprehensive evaluation of the code summarization models. To our knowledge, it is the first benchmark that can evaluate the model from multiple readability degrees using obfuscated code.

2 BACKGROUND AND PROBLEM FORMULATION

This section introduces the literature on code summarization and motivates our research questions.

2.1 Code Summarization

Code summarization refers to generating natural language comments for a given code snippet [15]. It is usually formulated as a machine translation task and is modeled by the standard encoder-decoder framework [15]. Given the input code $x = [x_1,...,x_N]$ with N tokens, a neural network is trained to generate the corresponding comment $y = [y_1,...,y_T]$ token by token based on the encoded vectors. The model can be implemented as a Transformer stacked by L self-attention layers [36]. Recently, large language models such as CodeLlama [26] have pushed the performance of code summarization to soaring heights.

2.2 Evaluation for Code Summarization Models

Code summarization can generally be considered a translation task. Therefore, the current evaluation methodology adheres to the conventional evaluation paradigm for translation: given code in a test set, the model generates natural language summaries, which are then compared against the ground-truth references in the test set. Performance is measured using holistic metrics such as BLEU [20], BERTScore [43], and ROUGE [19], which account for text similarities.

Although widely used, the conventional evaluation methodology has its limitations. Previous studies have shown that the performance of code summarization models is significantly influenced by the quality of the dataset [31]. Denoising the dataset by removing noisy instances can lead to substantial improvements in the models' capabilities [38].

2.3 Research Questions

Our hypothesis is that the readability of code could have effect to the comprehension by large language models. This leads to several research questions:

- **RQ1:** *How effective are state-of-the-art code summarization models on poor-readability code?* We investigate whether poor readability has a significant impact on the performance of code summarization models. We choose a code dataset that has been manually labeled with readability scores [3, 29]. We partition the dataset into high- and low-readability groups and compare the performance of code summarization models between the two groups.
- **RQ2:** *How do changes in code readability affect the performance of code summarization models?* Having learned that existing code summarization models are vulnerable to poor readability code, we want to know how different readability factors impact the model's performance. We perform an in-depth analysis of the robustness of code comprehension models in terms of different readability factors, including: 1) Semantic cues that characterize the intent of the code, such as function signatures, and variable names. Previous research has shown that function names play a dominant role in code comprehension [6, 44], and 2) Structural features such as keywords, operators, and syntactic symbols. We extend the primary benchmarks into seven obfuscated datasets via code perturbations. We compare the performance and robustness along the seven dimensionalities.
- **RQ3:** *How to fully assess the capability and robustness of a code summarization model?* Having identified that current code language models are sensitive to semantic features of code, we encounter a new challenge: the sensitive features may obscure the effects of non-sensitive ones when testing a model's capabilities. For instance, if a model heavily relies on semantic features such as identifiers, it becomes difficult to discern the impact of syntactic perturbations when the input code is rich in semantic information. To address this issue and evaluate models' ability to recognize low-sensitivity aspects of code, we propose cross-obfuscation benchmarks. These benchmarks aim to reduce the dominance of highly sensitive aspects by perturbing the original datasets using a combination of two perturbation rules: one from semantic perturbation and one from syntactic perturbation. By summing up the results of both robust evaluation and cross-obfuscation evaluation, we can gain more insights into the model's ability including performance, capacity, and robustness.

3 RQ1: EFFECT OF LOW-READABILITY CODE

In this section, we investigate the overall difference between low- and high-readability code in real-world code summarization.

3.1 Methodology

We test LLMs on a readability-annotated code dataset introduced by previous work [3, 29]. The dataset consists of 200 Java snippets, each assigned a readability score (ranging from 1 to 5) manually rated by nine developers. Out of the 200 snippets, 45 include corresponding documentation comments that serve as reference summaries for evaluating code summarization models. We divided the 45 snippets into two groups based on their readability scores: a high-readability group comprising 23 samples with the highest scores and a low-readability group consisting of 22 samples with the lowest readability scores. For each snippet, the readability score was calculated as the average of all developers' assessments.

We use GPT-4o[1] as the baseline model. GPT-4o is a cutting-edge commercial LLM that offers an accessible interface (i.e., prompting through a web page), without requiring extensive technical modifications. This allows us to capture the general distinctions between low- and high-readability code in real-world code summarization. Following previous work on ChatGPT for code summarization [33], we use the prompt "Please generate a short comment in one sentence for the following function:<code>" to elicit GPT-4o to output the code summary.

3.2 Evaluation Metrics

We evaluate the model performance using two metrics, including the smoothed BLEU-4 score [20] and the BERTScore [43], which are the most widely used metrics for shot-text summarization [12, 22, 35].

BLEU [20] is a standard algorithm for evaluating machine translations. It is computed as the ratio of n-gram tokens from the candidates that also appear in the references, with a penalty for too short length.

BERTScore [43] evaluates the sentence-level similarity of the candidate sentence to its reference by leveraging BERT's contextual embeddings, thereby addressing the shortcomings of earlier approaches and achieving a higher correlation with human judgment [43]. BERTScore has demonstrated exceptional performance across various NLP tasks [24, 35], establishing a new state-of-the-art benchmark in the field. BERTScores often lie in a small range. Following the general practice, we rescale them with baselines to fall in [0, 1][2].

3.3 Results

Table 1 summarizes the performance of GPT-4o in both readability groups. There is a clear discrepancy in performance between the two groups. The higher the readability, the greater the BLEU and BERTScore that the model achieves. We particularly notice that the BLEU score drops from 8.12 to 6.32 in the high- and low-readability groups, respectively. Additionally, all comparisons gain statistical significance. The results indicate that readability has a strong impact on the code comprehension performance by language models.

[1]https://chatgpt.com
[2]Rescaling BERTScore with baselines

Table 1: Code summarization performance of GPT-4o on codes with different readability.

Group	Readability	BLEU	BERTScore
low-readability	3.44	6.32	18.25
high-readability	4.28	**8.12**	**19.78**
p-value*	<0.0001	<0.01	<0.04 (0.0383)

**p-value is calculated with pairwise 2-sample Wilcoxon Signed rank test between the two groups. We repeat the comparison experiments 6 times.*

Answer to RQ1: Readability has a strong impact on the code summarization performance. Existing code summarization models exhibit a strong vulnerability to poor-readability code.

4 RQ2: FINE-GRAINED SENSITIVITY ON VARYING READABILITY FACTORS

In the previous section, we examined the overall impact of readability in code summarization models. To further understand the models' robustness on varying readability, we perform an in-depth analysis of the specific code features to which these models are sensitive.

4.1 Methodology

To gain more insights into the sensitivity of code summarization models across different readability features, we extend existing code summarization benchmarks to varying readability through code obfuscation. Code obfuscation involves perturbating source code in a way that makes it difficult for humans to understand while retaining its original functionality and informational content. It is widely used in software development for various purposes, such as protecting intellectual property, preventing reverse engineering, and enhancing security. By incorporating obfuscated code into existing benchmarks, we aim to evaluate how robust code summarization models are on varying readability while maintaining their ability in code comprehension.

4.1.1 Primary Benchmarks. We initially select three primary benchmarks for evaluating code summarization models, including TL-CodeSum[3] [13], DeepCom[4] [12], and CodeSearchNet[5] [14].

TL-CodeSum [13] released a dataset that includes 87,136 (function, summary) pairs extracted from Java projects created from 2015 to 2016 with at least 20 stars.

DeepCom [12] released a dataset that includes 588,108 Java methods with documentation. The dataset was originally collected from 9,714 GitHub projects. It takes the first sentence of the documentation comment as the summary of each Java method.

CodeSearchNet [14] is a well-formatted code language dataset. The dataset involves a large number of functions along with their documentation or comments written in Go, Java, JavaScript, PHP, Python, and Ruby. Our study requires extensive perturbations on the original code. Perturbating all languages would be an enormous workload. We selected Java, Python, and Go, the most popular languages from CodeSearchNet for efficient analysis.

[3]https://github.com/xing-hu/TL-CodeSum
[4]https://github.com/xing-hu/DeepCom
[5]https://github.com/github/CodeSearchNet

Table 2: Number of test functions on the original dataset in terms of different languages.

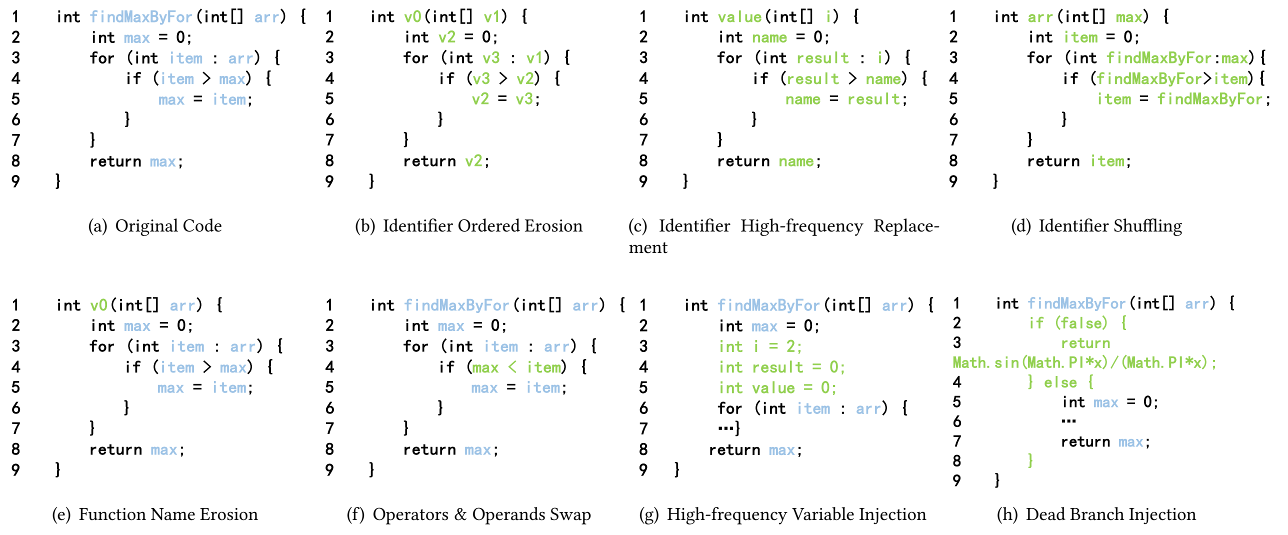
Dataset	Language	# of Functions
TL-CodeSum	Java	8,714
DeepCom	Java	58,811
CodeSearchNet	Java	10,955
	Python	14,918
	Go	8,122

The statistics of the benchmarks are presented in Table 2. As an evaluation benchmark, we only use their test sets in our experiments.

4.1.2 Creating Obfuscated Benchmarks. Based on the primary benchmarks, we construct obfuscated benchmarks that are more challenging for models to summarize. Previous research has shown that code readability relies on two channels of features: syntactic and textual [27, 29]. The former concerns more about the structure while the latter concerns the semantic cues (e.g., identifiers, keywords, and comments) used by humans for code comprehension. We hereby construct obfuscated code by perturbing both the semantics and syntax of the primary datasets. For semantic perturbation, we perturb the identifiers and function names of the original code. For syntactic perturbation, we use perturb operators, condition statements, and variable declarations. We did not perturb the loop statements like other work [4] because the *for* and *while* loops cannot completely be converted to each other across several programming languages. Besides, they do not account for a high proportion of the primary datasets. In addition, we consider the inclusiveness of the evaluation system for programming languages, for example, there is no "while" in Go.

Overall, we employ seven rules for constructing obfuscated code:

- **Identifier Ordered Erosion** (IOE) [7, 16, 40], replace identifiers in the code with ordered symbols such as "v0" and "v1", and modify the replaced identifier in the summary accordingly. This perturbation rule aims to erode semantic cues in the original identifiers.
- **Identifier High-frequency Replacement** (IHR) derived from identifier mangling [2, 16], replace identifiers with frequent tokens, and modify the replaced identifier in the corresponding summary. Compared to IOE, IHR produces code that is closer to the original data, as it only introduces frequent tokens in the training corpus instead of rare symbols adopted by IOE.
- **Identifier Shuffling** (IS) [2, 16], randomly shuffles identifiers within the same code snippets and replaces the new identifier in the summary accordingly. This rule perturbs the code sequence while preserving the semantic cues of the original code.
- **Function Name Erosion** (FNE) [5, 9, 10], replaces function names with symbols such as "v0", "v1" and modify all their occurrence in the corresponding summary. Function names



```
int findMaxByFor(int[] arr) {
    int max = 0;
    for (int item : arr) {
        if (item > max) {
            max = item;
        }
    }
    return max;
}
```

(a) Original Code

```
int v0(int[] v1) {
    int v2 = 0;
    for (int v3 : v1) {
        if (v3 > v2) {
            v2 = v3;
        }
    }
    return v2;
}
```

(b) Identifier Ordered Erosion

```
int value(int[] i) {
    int name = 0;
    for (int result : i) {
        if (result > name) {
            name = result;
        }
    }
    return name;
}
```

(c) Identifier High-frequency Replacement

```
int arr(int[] max) {
    int item = 0;
    for (int findMaxByFor:max){
        if (findMaxByFor>item){
            item = findMaxByFor;
        }
    }
    return item;
}
```

(d) Identifier Shuffling

```
int v0(int[] arr) {
    int max = 0;
    for (int item : arr) {
        if (item > max) {
            max = item;
        }
    }
    return max;
}
```

(e) Function Name Erosion

```
int findMaxByFor(int[] arr) {
    int max = 0;
    for (int item : arr) {
        if (max < item) {
            max = item;
        }
    }
    return max;
}
```

(f) Operators & Operands Swap

```
int findMaxByFor(int[] arr) {
    int max = 0;
    int i = 2;
    int result = 0;
    int value = 0;
    for (int item : arr) {
    ...}
   return max;
}
```

(g) High-frequency Variable Injection

```
int findMaxByFor(int[] arr) {
    if (false) {
        return
Math.sin(Math.PI*x)/(Math.PI*x);
    } else {
        int max = 0;
        ...
        return max;
    }
}
```

(h) Dead Branch Injection

Figure 2: Illustration of the seven perturbation rules. The perturbed parts are marked in green

have always been considered the most critical informative in code comprehension [6, 44]. By erasing semantics in the function name, this rule encourages the model to learn the importance of different identifiers in code.

- **Operators and Operands Swap** (OOS) [4], inverses the operators of binary and logical operations in the code and swap the corresponding operands to maintain the semantic. For inequalities in logical operations, we swap "<" and ">" (including "<=" and ">=") and their corresponding operands. For operators such as "+", "*", "==", "!=" or "<>", we only swap the operands.
- **Dead Branch Injection** (DBI) derived from code injection techniques [4, 7], this rule inserts condition statements into the code that do not affect the original program execution. Specifically, we inject a dead branch condition statement at the beginning of the function body, placing the original function body into the true branch and inserting unrelated code into the false branch. Condition statements are a fundamental code syntax feature, and this rule helps in evaluating the impact of such perturbations on the model.
- **High-frequency Variable Injection** (HVI) derived from code injection techniques [4, 7], this rule inserts variable declarations into the code that do not affect the original program execution.

Figure 2 illustrates examples of the seven perturbation rules. We implemented these perturbations by adopting existing obfuscation techniques. For example, we adopt NatGen [4] for DBI and HVI. The scripts for the seven perturbations can be found in our source code.

It is important to note that to maintain the readability of the perturbed code and prevent semantic deviations from the original code, we treat function names, parameters, and local variable names as identifiers. This implies that external APIs within the function body and global variables declared outside the function body will be retained.

Finally, by applying the seven perturbation rules to the original benchmarks, we obtain seven obfuscated benchmarks that reflect varying readability features. We proceed to evaluate code summarization models on these benchmarks. By analyzing the sensitivity of specific perturbations, we can identify the most crucial features of the code that influence the model's performance. For instance, if certain perturbations notably affect the model's performance, it suggests that the corresponding features (e.g., function name) are critical to the code summarization models. This approach also helps determine whether the model is more sensitive toward semantics or syntax within the code. By combining different perturbation rules, we can also analyze the more comprehensive capability of code language models.

4.2 Studied Models

In RQ2, we aim to perform a more in-depth analysis of the subtle differences across different readability dimensions and obfuscation rules. We opt for open-source models with various sizes that allow for both fine-tuning and zero-shot modes. This choice enables us to design more sophisticated prompts and perform a detailed, granular, and computationally efficient analysis of large-scale code data. We select three code comprehension models as our investigated models: CodeBERT, CodeT5, and CodeLlama. They stand as the most representative models for code summarization.

CodeBERT [8] is an encoder-only model based on RoBERTa [21]. The model has been pre-trained on both natural and programming languages using two self-supervised objectives, namely, masked language model and replaced token detection.

CodeT5 [39] is a code language model based on the encoder-decoder architecture. CodeT5 extends T5 by adding pre-training tasks to capture identifier semantics. Moreover, a bimodal dual generation task is proposed to enhance the decoder for generation tasks. We directly run the pre-trained checkpoint[6] for code summarization.

CodeLlama-7B [26] stands as the state-of-the-art LLM for code generation and comprehension. It is built on top of Llama 2 [34] and was pre-trained in Python. CodeLlama can generate both code and natural language about code. We generate summaries for Python using the infilling method recommended in the paper [26]. The infilling method replaces the documentation comment of the function body with a special tag `FILL_ME`, and takes it as input to the model[7]. The model will generate summaries to fill in the special tag. Since the content generated by CodeLlama is too long, we only select the first line as the final summary. CodeLlama has not been pre-trained in Go and Java. There is no officially recommended inference method in these languages. Therefore, for Go and Java, we generate summaries using the method in bigcode-evaluation-harness[8] which results in reasonable performance. Specifically, we append a prompt `Please fill this sentence "The goal of this function is to" in 12 words:` at the end of the code to guide the model to output the code summary.

We trained all code language models based on the released pre-trained checkpoint on huggingface[9][10][11]. All parameters were consistent with the open-source documents provided in the original papers of each model. We trained and evaluated all models on a GPU machine with Nivida Tesla V100. The hyperparameters and specifications for all language models are provided in Table 3.

Table 3: Hyperparameters of baseline models.

	CodeBERT	CodeT5	CodeLlama
Transformer layers	12	24	32
Max seq length	512	512	2048
Embedding size	768	768	4096
Attention head	12	12	32
Vocabulary size	50,265	32,100	32,000
# of parameters	125M	220M	7B

4.3 Results

Table 4 compares the results of three code language models on both the primary and obfuscated test sets. Generally, all models exhibit a decline in BLEU and BERTScore on the perturbed datasets. The findings from BERTScore closely align with those from BLEU. The results confirm our conclusion that existing code summarization models are vulnerable to poor code readability.

Comparing the perturbated datasets. Figure 1 illustrates the robustness of CodeT5 to different readability factors in seven obfuscated test sets. Across all perturbations, IOE and IHR have the most significant impact on the model performance. For example, IOE drops the BLEU scores obtained by CodeT5 by 5.15 and 6.48 in Python and Java, respectively. Codellama witnesses a BLEU drop by 10.88 in Python. IHR decreases the BLEU score obtained by CodeBERT by 5.41 and 5.68 in Python and Go, respectively. A similar trend can be observed by CodeLlama, with BLEU scores dropped by 0.93 and 2.48 in Java and Go, respectively.

FNE also plays a role in code summarization, particularly for smaller models. For example, FNE decreases the BLEU scores by 5.49 and 3.48 for CodeBERT and CodeT5 respectively in the Go language.

Overall, the results indicate that meaningful identifiers are critical for language models in code comprehension.

Comparing semantic and syntactic perturbations. To gain a more in-depth comparison between semantic and syntactic perturbations, we summarize the decrease of BLEU scores in Table 5. The results show that all code language models are more sensitive to semantic perturbations than to syntactic ones. According to the significance test, the decreases are significant for most semantic perturbations ($p<0.001$) while non-significant for most syntactic perturbations ($p>0.5$). The results suggest that code language models pay more attention to semantic features in the source code, such as function and variable names. On the other hand, simple structural perturbations do not significantly affect LLMs in code comprehension.

We particularly notice that the impact of syntactic perturbed datasets is overshadowed by semantic perturbations. For instance, semantic perturbations decrease the BLEU scores by 33.23% in Python code summarization using CodeLlama, while syntactic perturbations only result in a 0.95% decrease with the same model and language.

Meanwhile, most training tasks of code language models lack explicit syntactic supervision. This strengthens the connection between meaningful identifiers and summaries, thereby reducing the model's robustness against semantic perturbations.

The results suggest the need for a more intensive evaluation of syntactic perturbations to gain a deeper understanding of model performance.

Comparing the code language models. We provide our comprehensive analysis based on three evaluation aspects: performance, capability, and robustness. In terms of performance (Table 4, CodeLlama outperforms other baseline models on the primary test sets, though it shows suboptimal scores on Go and Java which have not been used for pre-training the model. From the model's capability, all three baseline models tend to focus on the perturbation of semantics over syntax. This is particularly evident in larger models such as CodeLlama. For robustness, the results presented in Table 5 suggest that CodeBERT is robust to strong perturbations in Python, while in Go and Java, CodeLlama exhibits more robustness. Overall, the higher the score a model achieves, the more sensitive it is to semantic perturbations.

4.4 Qualitative Analysis

In addition to the quantitative analysis, we also provide a number of cases. Figure 3 presents three cases of code summarization by CodeT5. We compare the human-written references to the model-generated summaries with and without FNE perturbation, respectively. FNE (replacing function names with `v0`) aims to reduce the

[6]CodeT5 generation script.
[7]CodeLlama infilling method.
[8]Prompts for code-to-text task with large model.
[9]https://huggingface.co/microsoft/codebert-base
[10]https://huggingface.co/Salesforce/codet5-base
[11]https://huggingface.co/codellama/CodeLlama-7b-hf

Table 4: Evaluation results on the obfuscated datasets (BL=BLEU, BS=BERTScore). The values in italic indicate nonsignificant decrease (p>0.05)

Dataset	CodeBERT						CodeT5						CodeLlama					
	Python		Go		Java		Python		Go		Java		Python		Go		Java	
	BL	BS	BL	BS	BL	BS	BL	BS	BL	BS	BL	BS	BL	BS	BL	BS	BL	BS
Primary																		
	17.95	29.64	17.78	40.11	18.62	31.92	20.38	34.41	19.67	43.18	20.66	35.35	22.03	38.91	12.78	26.30	15.11	33.68
Semantic Perturb.																		
IOE	13.89	19.52	**11.02**	**16.26**	13.85	21.53	**15.23**	**22.05**	**16.90**	37.09	**14.18**	**20.90**	**11.15**	**13.27**	*12.75*	*26.26*	13.34	29.40
IS	14.70	22.82	13.07	23.28	15.42	25.50	16.50	26.97	17.87	37.64	15.88	26.51	17.30	30.03	*12.40*	*25.54*	13.95	31.43
IHR	**12.54**	**17.53**	12.10	22.38	**12.84**	**19.61**	15.84	25.03	16.96	**36.26**	14.82	24.65	11.86	20.02	**11.85**	**24.01**	**12.63**	**27.85**
FNE	14.74	22.63	12.95	21.48	15.40	25.17	17.05	26.99	16.19	35.37	15.72	22.60	18.54	31.78	*12.42*	25.18	14.17	31.28
Syntactic Perturb.																		
OOS	*17.94*	*29.63*	*17.79*	*40.14*	*18.61*	*31.90*	19.34	33.43	*19.71*	*43.27*	*20.68*	*35.45*	*22.08*	*38.86*	*12.77*	*26.29*	*15.08*	*33.70*
HVI	17.53	28.69	*17.75*	*40.08*	18.15	30.99	19.32	33.19	*19.62*	*43.00*	*20.65*	*35.25*	*21.69*	*38.53*	*12.88*	*26.46*	*15.22*	*33.67*
DBI	17.34	28.27	*17.87*	*40.40*	*18.26*	*31.41*	18.77	31.61	*19.15*	42.51	*20.25*	*34.87*	*21.69*	*38.49*	*12.95*	*26.28*	*14.95*	*33.06*

*p-value is calculated with pairwise 2-sample Wilcoxon Signed rank test between the primary and each perturbed dataset.

Table 5: The decrease of BLEU scores with their relative percentage for each robustness aspect.

Lang	Model	Semantic	Syntactic	Overall
Py	**CodeBERT**	**3.98/22.17%**	0.35/1.95%	**4.33/24.12%**
	CodeT5	4.22/20.71%	1.24/6.08%	5.46/26.79%
	CodeLlama	7.32/33.23%	**0.21/0.95%**	7.53/34.18%
Go	**CodeBERT**	5.49/30.88%	-0.02/-0.11%	5.47/30.76%
	CodeT5	2.69/13.68%	0.18/0.92%	2.87/14.59%
	CodeLlama	**0.42/3.29%**	**-0.09/-0.70%**	**0.33/2.58%**
Java	**CodeBERT**	4.24/22.77%	0.28/1.50%	4.52/24.27%
	CodeT5	5.51/26.67%	0.13/0.63%	5.64/27.30%
	CodeLlama	**1.59/10.52%**	**0.03/0.20%**	**1.62/10.72%**

semantic hints that the model can obtain from the function name. However, when the function name is eroded, the summary generated by CodeT5 becomes confusing because it still tends to use the information of `v0` for reasoning instead of summarizing the information in the code body. We also found this phenomenon with other semantic perturbations. Although these perturbations do not change the functionality of the program, they hamper the understanding of the overall semantics.

Answer to RQ2: The current code summarization models predominantly depend on semantic cues within the code such as variable names, often neglecting the syntactic features such as operators and branches. Existing benchmarks are biased toward evaluating semantic features, thereby overlooking the models' ability to understand syntactic features.

5 RQ3: ENHANCED ROBUSTNESS EVALUATION THROUGH CROSS-PERTURBATION

5.1 Methodology

Recognizing that models exhibit heightened awareness of semantic cues, we embark on an in-depth examination of their capability to comprehend code syntax. To achieve this, we introduce a novel cross-perturbation evaluation methodology aimed at minimizing the impact of semantic perturbations on the original code while intensifying the syntactic features. Figure 4 shows our proposed evaluation workflows, we construct balance-sensitivity datasets by combining two distinct sets of perturbation rules: one focusing on semantic perturbations and the other on syntactic perturbations. Specifically, we select semantic perturbation rules that have the maximal impact on the specific language and model. These chosen semantic perturbation rules are then paired with one of the syntax perturbation rules. We use this methodology to create a new benchmark named PoorCodeSumEval. We evaluate model performance on the new benchmark.

5.2 Results

Table 6 presents the results of the cross-perturbation evaluation in the new benchmark. Rows under "Semantic Perturb." represent the selected semantic perturbation rules that have the maximal impact on the performance on the primary benchmark. Rows with "Cross Perturb." show the results of cross perturbation when the chosen semantic perturbation is paired with each syntax perturbation. The results show that when the sensitive semantic aspect is eliminated from the code, the effect of syntactic perturbation begins to be evident. We can draw some conclusions from the new perturbation results.

Comparing the syntactic perturbations. The results indicate that all models exhibit significant sensitivity to perturbations involving *dead branch insertion* (DBI) (p<0.0001). *High-frequency variable*

Table 6: BLEU scores on the cross-obfuscated datasets. The semantic perturbation chooses the one with the greatest impact on the model in each programming language. The values in italic indicate nonsignificant decrease (p>0.05)

Dataset	CodeBERT			CodeT5			CodeLlama		
	Python	Go	Java	Python	Go	Java	Python	Go	Java
Semantic Perturb.	**IHR**	**IOE**	**IHR**	**IOE**	**FNE**	**IOE**	**IOE**	**IHR**	**IHR**
primary	12.54	11.02	12.84	15.23	16.19	14.18	11.15	11.85	12.63
Cross Perturb.									
Semantic × OOS	*12.52*	*10.93*	*12.87*	14.21	*16.25*	*14.34*	*11.07*	*11.64*	*12.50*
Semantic × HVI	11.57	*10.81*	12.16	13.75	*15.75*	13.58	8.85	*11.82*	*12.51*
Semantic × DBI	**11.49**	*10.74*	**12.13**	**12.70**	**14.58**	**13.31**	**8.15**	*11.41*	**12.07**
average	11.83	10.83	12.39	13.55	15.53	13.74	9.36	11.62	12.36

*p-value is calculated with pairwise 2-sample Wilcoxon Signed rank test between the primary and each perturbed dataset.

injection (HVI) also has an impact on model performance in some models and languages. We particularly notice that Semantic×OOS exhibits nonsignificant decrease (p>0.5). The results indicate that *operand swap* does not affect the readability significantly whereas *dead branch injection* is the most significant way for structural obfuscation.

Unlike syntactic perturbed datasets used for robustness evaluation, cross-perturbed datasets offer a more explicit revelation of the model's ability on low-sensitive aspects. This is achieved by minimizing the impact of high-sensitivity code aspects, providing a clearer understanding of the model's performance across various dimensions.

Comparing different programming languages. Among the three programming languages, Python exhibits the highest sensitivity to syntactic perturbation. This sensitivity may stem from Python's syntactic flexibility, causing code models to overly prioritize semantic information and making it challenging to differentiate syntactic perturbations.

Answer to RQ3: By eroding the semantic cues while intensifying the syntactic features, PoorCodeSumEval can more comprehensively assess the ability of code language models in code summarization.

6 DISCUSSION

6.1 Robustness and capability of code summarization models

Our experiments reveal a notable distinction between a model's capability and robustness. A model that demonstrates robustness to perturbations of specific features does not necessarily imply its proficiency in capturing those features. For example, our findings demonstrate that state-of-the-art code language models excel in capturing semantic features rather than syntax. Hence the impact of syntax perturbations on these models is marginal. In that sense, the model is robust to syntax perturbations, but this robustness does not inherently imply a strong ability to capture syntax features. The high evaluation score observed in metrics is a reflection of the model's proficiency in capturing semantic features, which may inadvertently overshadow its performance in capturing syntax features.

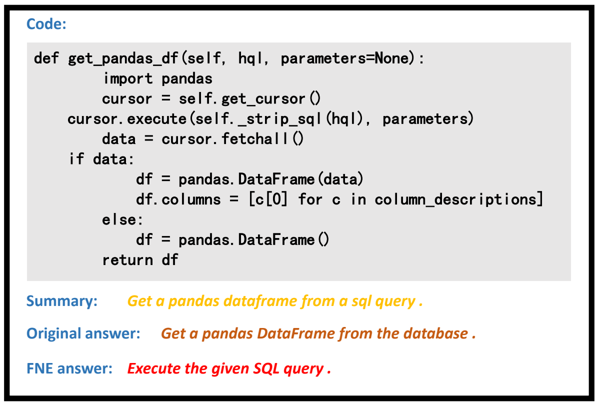
6.2 Why are code language models sensitive to identifier perturbations?

State-of-the-art code language models have been pre-trained on large-scale code corpora. These pre-training objectives force the models to focus on semantic cues (such as identifiers) that appear in both natural and programming languages. In other words, the performance of PLMs depends on the readability of the code. In practice, however, the readability of code cannot be guaranteed, which means that the model that achieves high evaluation scores on the original benchmarks may fail to comprehend code with poor readability in real datasets. This is particularly evident in larger language models such as CodeLlama, which have been pre-trained on large-scale standardized code.

6.3 Lessons Learned for Evaluating Code Language Models

The ability of code LMs has been a growing concern, especially when large language models (LLMs) have been widely applied for SE tasks. As LLMs typically demonstrate strengths from different perspectives, a comprehensive assessment of the model's ability has become a compelling need. Most existing benchmarks focus on correctness while overshadowing the robustness of models to varying code readability. When investigating the current code summarization models (as shown in Table 4), we notice that benchmarks with different obfuscation rules exhibit the different abilities of code LMs, thereby giving a comprehensive assessment of model performance. Therefore, providing code with different formats while evaluating code summarization models is recommended, rather than directly using existing datasets. Specifically, our study led to the following lessons learned from the quality assessment on benchmark datasets, for future code summarization researchers.

- Try to normalize identifiers and structures to conventional forms to mitigate the influence of code irregularities on the generated summaries.
- Try to erose semantic cues (e.g., identifiers, function names) to enhance the evaluation of syntactic comprehension by code language models.


Code:

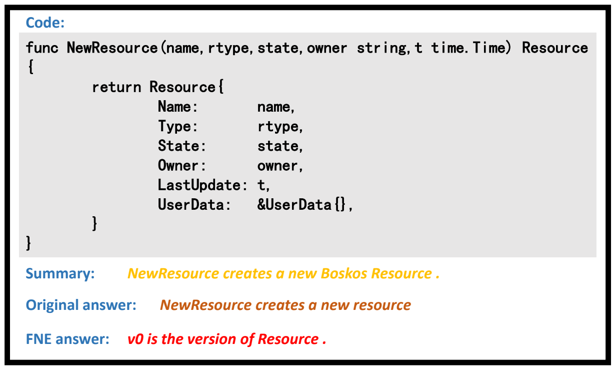
```
def get_pandas_df(self, hql, parameters=None):
        import pandas
        cursor = self.get_cursor()
    cursor.execute(self._strip_sql(hql), parameters)
        data = cursor.fetchall()
    if data:
            df = pandas.DataFrame(data)
            df.columns = [c[0] for c in column_descriptions]
        else:
            df = pandas.DataFrame()
        return df
```

Summary: *Get a pandas dataframe from a sql query .*

Original answer: *Get a pandas DataFrame from the database .*

FNE answer: ***Execute the given SQL query .***

(a) A case in Python.


Code:

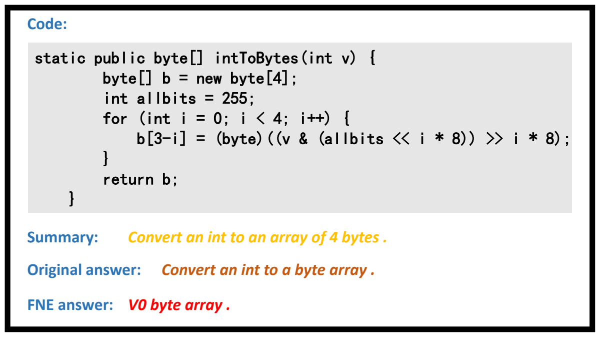
```
func NewResource(name, rtype, state, owner string, t time.Time) Resource
{
        return Resource{
                Name:       name,
                Type:       rtype,
                State:      state,
                Owner:      owner,
                LastUpdate: t,
                UserData:   &UserData{},
        }
}
```

Summary: *NewResource creates a new Boskos Resource .*

Original answer: *NewResource creates a new resource*

FNE answer: ***v0 is the version of Resource .***

(b) A case in Go.


Code:

```
static public byte[] intToBytes(int v) {
        byte[] b = new byte[4];
        int allbits = 255;
        for (int i = 0; i < 4; i++) {
            b[3-i] = (byte)((v & (allbits << i * 8)) >> i * 8);
        }
        return b;
    }
```

Summary: *Convert an int to an array of 4 bytes .*

Original answer: *Convert an int to a byte array .*

FNE answer: ***V0 byte array .***

(c) A case in Java.

Figure 3: Summaries generated by CodeT5 with and without FNE perturbation.

6.4 Implications for Research and Practice

Need for research on more readability factors In this study, we investigate the robustness of code summarization models through seven obfuscation rules in code summarization benchmarks. We believe that other characteristics of code could also reflect the readability of code. (1) **Code length**: whether longer code hinders models from comprehending source code. A more rigorous methodology could be separating code into groups with distinct characteristics, such as long and short code, and analyzing each group individually. (2) **Code smell**: more broadly, we could consider code smell as a feature of low readability. (3) **Inclusion of line comments**: Previous work has shown that high-quality line comments can improve the readability of the code [29]. Low-quality comments, on the other hand, may exhibit issues such as incoherent statements, grammatical errors, or semantic mismatches with the code, thereby increasing misunderstandings and reducing the overall readability of the code. (4) **Inclusion of domain libraries**: third-party libraries are frequently called in functions. Providing the background knowledge of domain libraries will greatly help improve the readability of the code.

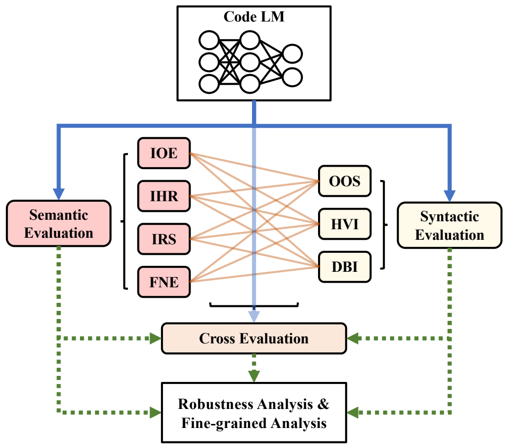


Figure 4: The proposed evaluation workflow of PoorCodeSumEval, the solid blue arrow indicates the corresponding evaluation of the model and the dashed green arrow indicates the transmission of evaluation results.

Need for creating more benchmarks Comprehensive evaluation of code language models has become a crucial objective in software engineering, especially in the era of large language models. Our study reveals that existing benchmarks are biased toward semantic features of code while neglecting the models' ability to comprehend syntax. This paper takes an initial step towards addressing the challenges of creating balanced benchmarks for code summarization models. Although we have proposed a new workflow to measure the model's ability to comprehend non-semantic aspects of code, our solutions might not be sufficient for other research tasks in software engineering that involve greater diversity, more characteristics, and larger volumes of data sources. Thus, we urge the research community to build benchmarks from multiple evaluation perspectives.

Need for research on more metrics Existing evaluation metrics such as BLEU and BERTScore primarily focus on the surface-level similarity between the generated summaries and the reference summaries. However, evaluating code summarization models from multiple perspectives is crucial. In this study, we introduce a novel evaluation workflow tailored to assess the robustness of code language models. We anticipate the emergence of more specialized evaluation metrics that target specific aspects of model performance, thereby enriching the evaluation process and enabling a more holistic assessment of code models.

Need for research on data normalization. Our study indicates that code language models strongly rely on semantic features such as identifiers. Meaningful and descriptive identifiers can greatly improve code readability [32], thereby facilitating semantic learning by code language models. On the other hand, programmers adhere to diverse programming conventions, leading to inconsistent formats and naming styles. This significantly hampers the learning of code semantics. Therefore, normalizing source code to the same format and style is crucial to enhance code comprehension. In this study, we investigate how code obfuscation impacts the performance of code language models. The widely-used datasets could contain other types of low-quality code that have not undergone cleaning and normalization processes. As LLMs typically require high-volume data, ensuring code quality at a large scale has become a compelling need.

7 THREATS TO VALIDITY

The use of synthetic poor-readability code. In large-scale experiments, we create low-readability code using synthetic perturbation rules instead of human-written low-readable code, which may reduce the validity of our conclusions in practice. To mitigate this threat, we consider two primary aspects of code readability, namely, semantic cues and syntactic rules. For each aspect, we choose the most representative obfuscation rules to mimic different cases of low readability in real-world programs. These perturbated benchmarks allow us to investigate the effectiveness of code language models in various readability conditions by controlling perturbation rules.

Using single holistic metrics to measure the comprehensive performance of models. Although our goal is to explore the models' performance across broader readability aspects, we employ holistic metrics such as BLEU and BERTScore in our study. We argue that these metrics can reflect comprehensive performance because we have expanded the primary test benchmarks to encompass various aspects through different obfuscation rules, generating multiple fine-grained results with a single metric. Consequently, the evaluated model can be analyzed by comparing its performance across different obfuscation datasets, unlike previous works that require comparisons of overall performance with other models.

8 RELATED WORK

8.1 Studies on Code Summarization Evaluation

Besides our work, there have been other works that study the evaluation of code summarization systems [30, 31]. For example, [30] carried out an in-depth analysis of the evaluation of the code summarization task. Their work focuses on several aspects of the evaluation process such as the data pre-processes, operation model used, characteristics of datasets, and evaluation metrics. Shi et al. [31] concern about the quality of the benchmark datasets built from real-world projects and study the noise in different benchmark datasets.

While these works study the quality of evaluation data, PoorCodeSumEval analyzes the robustness of code language models by obfuscating the test data under different readability factors. Additionally, PoorCodeSumEval offers a new evaluation workflow to comprehensively assess the model's performance in varying readability conditions. Table 7 compares PoorCodeSumEval with previous benchmarks on the code summarization task. As seen, PoorCodeSumEval provides a more rigorous evaluation for code summarization models: using test data of more challenging readability to analyze fine-grained aspects of the model including performance, capabilities, and robustness.

8.2 Studies on Code Readability

Code readability has long been a focal point in the study of software quality and maintenance [3, 18, 27–29]. Buse and Weimer propose an automated readability metric based on human annotations, showing strong correlations with software quality metrics such as code changes and defect reports. Their results suggest that readability can significantly impact software quality [3]. Scalabrino et al. advanced this field through a series of follow-up studies. They introduced textual features, including identifiers and comments, to complement structural ones in code readability models, significantly improving their accuracy [29]. Later, they continue to introduce additional textual features into code readability models. Their work demonstrated that a combined model of structural and textual features predicts readability and software issues more accurately [27]. They further investigated the correlation between various metrics and code understandability and found that traditional readability and complexity metrics do not effectively capture code understandability, highlighting the need for new metrics [28]. Johnson et al. conducted an experiment to assess readability rules. Their study found that minimizing nesting improved comprehension and bug-finding efficiency, whereas avoiding do-while loops had no significant effect [18].

Unlike previous works that investigate readability from human aspects, we study the effect of readability on code comprehension by code language models. Our work diverges from theirs by focusing on how readability affects code comprehension by language models, aiming to bridge the gap between human and machine readability studies.

8.3 Robustness of Code Language Models

Besides our work, there have been studies on the robustness of code language models. Wang et al. analyzed the robustness of code generation models [37] to noisy data samples [38]. Similarly, Yang et al. assessed and enhanced the robustness of programming language models in code translation tasks [41].

Some previous works aimed at enhancing the robustness of code models. Bielik et al. proposed a novel technique to train accurate and robust neural models of code, addressing key challenges specific to the domain of code, such as incomplete code snippets and the structured nature of programs [1]. Ramakrishnan et al. designed a robust-optimization objective training process to enhance the robustness of models against semantic-preserving code transformations [11]. Wang et al. employed noisy label learning (NLL) while

Table 7: Comparison of different benchmarks on the code summarization task. The capability refers to the ability of the model to capture semantics or syntax from code.

Benchmark	Multilingual	Multiple Datasets	Evaluation Aspects		
			Performance	Capability	Robustness
TL-CodeSum [13]	×	×	√	×	×
DeepCom [12]	×	×	√	×	×
CodeSearchNet [14]	√	×	√	×	×
CodeXGLUE [23]	√	×	√	×	×
XLCoST [46]	√	√	√	×	×
PoorCodeSumEval	√	√	√	√	√

training code models, achieving notable results, particularly for small models [38].

Some previous works exploit the vulnerabilities in code language models and attack them using adversarial samples. Yefet et al. presented the first approach for attacking code models using adversarial examples [42]. Jha et al. introduced CodeAttack, a black-box adversarial attack model that detects vulnerabilities in small pre-trained models across tasks [17]. Zhou et al. [45] studied the robustness of pre-trained code summarization models. They introduce ACCENT which generates adversarial samples to enhance model robustness through adversarial training.

Our paper presents several significant differences to related works: First, related studies were not designed upon LLMs. They were built upon Transformer without pre-training or solely small language models. Consequently, their findings may not be applicable to LLMs. By contrast, we are the first to analyze code summarization models on poor-readability code using LLMs. Second, previous works focus solely on identifier perturbation, whereas PoorCodeSumEval offers a comprehensive analysis across various readability dimensions and obfuscation rules, including both semantic and syntax. Finally, we propose a novel cross-obfuscation methodology to evaluate the LLM's ability in comprehending the insensitivity aspects of code.

9 CONCLUSION

In this paper, we propose PoorCodeSumEval, a robustness evaluation benchmark for code summarization tasks. PoorCodeSumEval modifies existing benchmarks to reflect varying degrees of readability through code obfuscation, thereby enabling a comprehensive evaluation of code summarization models on multiple aspects of model performance. Our experiments, conducted on three baseline models, reveal that existing models showcasing high BLEU and BERTScores exhibit high sensitivity to semantic features in code, particularly identifiers. Notably, larger models exhibit a more pronounced dependence on these semantic features, making them vulnerable to code with poor readability. Our work can inspire future research to evaluate more aspects of model capability, instead of relying on individual metrics and datasets.

DATA AVAILABILITY

We make PoorCodeSumEval, benchmark models, and code publicly available at https://github.com/ythere-y/PoorCodeSumEval

ACKNOWLEDGMENTS

This research is supported by National Key R&D Program of China (Grant No. 2023YFB4503802) and National Natural Science Foundation of China (Grant No. 62102244).

REFERENCES

[1] Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *International Conference on Machine Learning*. PMLR, 896–907.

[2] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.

[3] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on software engineering* 36, 4 (2009), 546–558.

[4] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 18–30.

[5] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[6] Xi Ding, Rui Peng, Xiangping Chen, Yuan Huang, Jing Bian, and Zibin Zheng. [n. d.]. Do Code Summarization Models Process Too Much Information? Function Signature May Be All What Is Needed. *ACM Transactions on Software Engineering and Methodology* ([n. d.]).

[7] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2021. Contrastive Learning for Source Code with Structural and Functional Properties. *arXiv preprint arXiv:2110.03868* (2021).

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[9] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. *arXiv preprint arXiv:1811.01824* (2018).

[10] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1667–1680.

[11] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 526–537.

[12] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.

[13] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. (2018).

[14] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[15] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2073–2083.

[16] Paras Jain and Ajay Jain. 2021. Contrastive Code Representation Learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.

[17] Akshita Jha and Chandan K Reddy. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 14892–14900.

[18] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. 2019. An empirical study assessing source code readability in comprehension. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 513–523.

[19] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[20] Chin-Yew Lin and Franz Josef Och. 2004. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. 501–507.

[21] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[22] Jiasen Lu, Dhruv Batra, Devi Parikh, and Stefan Lee. 2019. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. *Advances in neural information processing systems* 32 (2019).

[23] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[24] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PMLR, 8748–8763.

[25] Pooja Rani, Suada Abukar, Nataliia Stulova, Alexandre Bergel, and Oscar Nierstrasz. 2021. Do comments follow commenting conventions? a case study in java and python. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 165–169.

[26] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[27] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: How far are we?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 417–427.

[28] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 6 (2018), e1958.

[29] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2016. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[30] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*. 1597–1608.

[31] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–119.

[32] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 378–389. https://doi.org/10.1145/2568225.2568252

[33] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865* (2023).

[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[35] Inigo Jauregi Unanue, Jacob Parnell, and Massimo Piccardi. 2021. BERTTune: Fine-tuning neural machine translation with BERTScore. *arXiv preprint arXiv:2106.02208* (2021).

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[37] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 13818–13843.

[38] Wenhan Wang, Yanzhou Li, Anran Li, Jian Zhang, Wei Ma, and Yang Liu. 2024. An Empirical Study on Noisy Label Learning for Program Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[39] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. 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*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[40] Fan Yan and Ming Li. 2021. Towards Generating Summaries for Lexically Confusing Code through Code Erosion.. In *IJCAI*. 3721–3727.

[41] Guang Yang, Yu Zhou, Xiangyu Zhang, Xiang Chen, Tingting Han, and Taolue Chen. 2023. Assessing and Improving Syntactic Adversarial Robustness of Pre-trained Models for Code Translation. *arXiv preprint arXiv:2310.18587* (2023).

[42] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[43] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).

[44] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet Code Is Healthy: Simplifying Programs for Pre-Trained Models of Code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. ACM.

[45] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. 2022. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–30.

[46] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence. *arXiv preprint arXiv:2206.08474* (2022).