How Robust Are Code Summarization Models to Poor-Readability Code? Fine-grained Evaluation and Benchmark

Anonymous ACL submission

Abstract

Pre-trained language models such as CodeT5 have demonstrated substantial achievement in code comprehension. Despite the giant leap in model architectures and training processes, we find that the benchmarks used for evaluating code summarization tasks are confined to high-readability code, regardless of the popularity of obfuscated code in reality. As such, they are inadequate to demonstrate the fine-grained ability of models, particularly the robustness to varying readability degrees. In this paper, we introduce OR-CodeSum, a robust evaluation benchmark on code summarization tasks, including seven obfuscated datasets derived from existing datasets. OR-CodeSum innovatively introduces the construction rules of obfuscation code into the testing process, considering semantic, syntactic, and cross-obfuscation robustness of code summarization tasks. Our robustness evaluation reveals that the current code summarization models rely heavily on the readability of the code while not paying enough attention to the syntactic information. We believe OR-CodeSum[1] can help researchers obtain a more comprehensive and profound understanding of code summarization models, which facilitates the improvement of model performance.

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 large pre-trained models such as CodeBERT, CodeT5, and CodeLlama (Feng et al., 2020; Wang et al., 2021; Roziere et al., 2023).

[1]We make OR-CodeSum, benchmark models, and code publicly available.

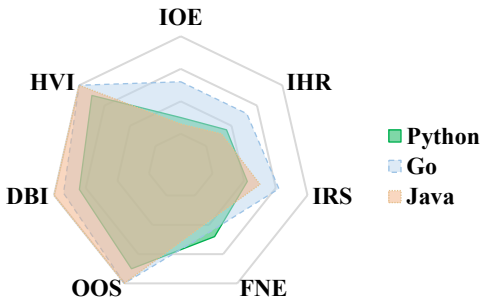


Figure 1: Fine-grained evaluation for CodeT5 across three programming languages. The seven dimensions represent seven perturbed datasets derived from existing datasets. Each value within the figure corresponds to the percentage of results obtained from the perturbed dataset in relation to the results from the original dataset.

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 (Rani et al., 2021; Shi et al., 2022b; Bielik and Vechev, 2020). 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 (Rani et al., 2021; Shi et al., 2022b; Bielik and Vechev,

2020). **Second**, and more critically, the current evaluation processes rely on single, coarse-grained metrics, offering only an overall performance assessment of neural network models on a specific task. Consequently, these metrics fail to provide a comprehensive and intuitive demonstration of the fine-grained capabilities of pre-trained code models, particularly the model's robustness to varying readability degrees.

In this paper, we present OR-CodeSum, a robust evaluation benchmark designed to assess the robustness of code summarization models. To provide a thorough evaluation of model robustness against perturbations, we expand three widely used datasets, namely TL-CodeSum (Hu et al., 2018b), Deepcom (Hu et al., 2018a), and CodeSearchNet (Husain et al., 2019), into seven finer-grained datasets using via obfuscated samples, i.e., obfuscated code that have the same semantics as the original code. We construct obfuscation samples according to seven perturbation rules that target both semantic and syntactic aspects of code: 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). Figure 1 illustrates the fine-grained performance of CodeT5 in our obfuscated test set.

We conduct a robustness analysis on three prominent code LMs, namely CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021), and CodeLlama (Roziere et al., 2023), using the obfuscated datasets. This analysis aims to gain insights into the model's sensitivity to various aspects. Subsequently, we create cross-perturbed datasets by minimizing the impact of sensitive aspects, allowing us to assess the comprehension ability of models on low-sensitive code aspects. Finally, we scrutinize the fine-grained performance of code summarization models using the results of both the robust evaluation and cross-evaluation on the reduced sensitivity datasets.

The primary contributions of this paper are summarized as follows:

- We introduce a new benchmark dataset for a fine-grained evaluation of the code summarization task across three programming languages. To the best of our knowledge, it is the first benchmark that can evaluate the model from performance, capability, and robustness by using obfuscated code.
- We create an evaluation workflow to assess the fine-grained capability of neural code summarization models, taking into account their semantic, syntactic, and joint robustness.
- We conduct experiments to analyze the fine-grained characteristics of state-of-the-art code language models.

2 Related Work

2.1 Code Benchmarks

Besides our work, there have been other benchmarks for code understanding and generation tasks (Lu et al., 2021; Zhu et al., 2022).

CodeXGLUE (Lu et al., 2021), a benchmark that provides a fine-grained taxonomy for a broad range of code comprehension and generation tasks by collecting and integrating the previously publicly used datasets. The benchmark involves 14 datasets, a collection of 10 different code comprehension and generation tasks, and a platform for model evaluation and comparison.

XLCoST (Zhu et al., 2022), a benchmark dataset for cross-lingual code intelligence. As it is always difficult to mine parallel code data, XLCoST provides fine-grained parallel data in 7 commonly used programming languages (C++, Java, Python, C#, Javascipt, PHP, C), and natural language (English).

Table 1 compares OR-CodeSum with previous code benchmarks on the code summarization task. As seen, OR-CodeSum 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.

2.2 Studies on Code Summarization Evaluation

Besides our work, there have been other works that study the evaluation of code summarization systems (Shi et al., 2022a,b). For example, (Shi et al., 2022a) carried out an in-depth analysis of the evaluation of the code summarization task. Their work focuses on several aspects in the evaluation process such as the data pre-processes, operation model used, characteristics of datasets, and evaluation metrics. Shi et al. (2022b) concern about the quality of the benchmark datasets built from real-world projects and study the noise in different benchmark datasets.

Benchmark	Multilingual	Multiple Datasets	Evaluation Aspects		
			Performance	Capability	Robustness
CodeXGLUE	√	×	√	×	×
XLCoST	√	√	√	×	×
OR-CodeSum	√	√	√	√	√

Table 1: 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.

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

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

While these works study the quality of evaluation data, OR-CodeSum aims at providing a fine-grained evaluation of the model by obfuscating the test data in different aspects and analyzing the capability of the tested model.

3 Dataset

In this section, we introduce our proposed dataset called OR-CodeSum. OR-CodeSum is built upon three previously publicly used datasets. to provide more fine-grained evaluations of code summarization models. We extend them into seven obfuscated datasets via code perturbations.

3.1 Primary Datasets

OR-CodeSum is built on three existing code summarization datasets, including TL-CodeSum [2](Hu et al., 2018b), DeepCom[3] (Hu et al., 2018a), and CodeSearchNet [4](Husain et al., 2019).

TL-CodeSum (Hu et al., 2018b) 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 (Hu et al., 2018a) 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 (Husain et al., 2019) 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. It was parsed using TreeSitter[5].

The statistics of the primary datasets are presented in Table 2. As an evaluation benchmark, OR-CodeSum only uses their test sets.

3.2 Obfuscation Datasets

To give a fine-grained evaluation of code summarization models, we extend the primary datasets with obfuscated samples. Obfuscation samples refer to samples that are different from the original samples but provide the same information for the code summarization model (Margatina et al., 2021; Jain and Jain, 2021; Chakraborty et al., 2022). They are often used to deceive deep learning models in the training stage and help improve the robustness of the model (Margatina et al., 2021).

In our work, we want obfuscated examples to contain the full aspects of the original code. Previous research has shown that source code involves two channels of information: formal and natural (Casalnuovo et al., 2020; Chakraborty et al., 2022). The formal channel concerns more about the syntax of code represented by abstract syntax trees (AST) and data flow diagrams (DFS). The natural channel, on the other hand, concerns the semantic features (e.g., identifiers, keywords, and comments) used by humans for code comprehension. As such, we construct obfuscation samples by perturbing both the semantics and syntax of the primary datasets. For semantic perturbation, we use four rules to perturb the identifiers and function names of the original code. For syntactic perturbation, we use three rules to perturb operators, condition statements, and vari-

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

able declaration statements. We did not perturb the loop statements like other work (Chakraborty et al., 2022) 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 samples:

- **Identifier Ordered Erosion** (IOE) (Yan and Li, 2021; Jain and Jain, 2021; Ding et al., 2021), 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 the original identifiers with the ordered symbols that are relatively rare to model.
- **Identifier High-frequency Replacement** (IHR) derived from Identifier mangling (Jain and Jain, 2021; Bui et al., 2021), replace identifiers with frequent tokens, and modify the replaced identifier in the corresponding summary. Compared to IOE, data after IHR is closer to the original data, as it only introduces frequent tokens in the training corpus instead of rare symbols adopted by IOE.
- **Identifier Random Shuffling** (IRS) (Jain and Jain, 2021; Bui et al., 2021) , randomly shuffle identifiers within the same code snippets and modify the replaced identifier in the summary accordingly. This rule perturbs the code sequence from the perspective of preserving the semantics of the original code.
- **Function Name Erosion** (FNE) considering the importance of function name for code snippets (Fernandes et al., 2018; David et al., 2020; He et al., 2018), replace function names with special tokens, such as "v0", "v1" and modify all their occurrence in the corresponding summary. The function name is always the most informative in the source code. Similar to IOE, this rule, by erasing semantics in the function name, can encourage the model to learn the importance of different identifiers in code.
- **Operators and Operands Swap** (OOS) (Chakraborty et al., 2022), as logical and numerical operations play an important role in code syntax and style, we inverse the operators of binary and logical operations in the code and swap the corresponding operands to keep the same semantic. For inequalities in logical operations, we replace "<" and ">" with each other (including "<=" and ">=") and swap the corresponding operands. For operators such as "+", "*", "==", "!=" or "<>", we only swap the operands.
- **Dead Branch Injection** (DBI) derived from code injection (Ding et al., 2021; Chakraborty et al., 2022), condition statement is the basis of code syntax, we insert branch statements in the code that will not affect the original program execution process. We inject a dead branch condition statement at the beginning of the function body and put the original function body into the true branch while inserting unrelated code into the false branch.
- **High-frequency Variable Injection** (HVI) derived from code injection (Ding et al., 2021; Chakraborty et al., 2022), we insert variable declaration statements in the code that will not affect the original program execution process.

Figure 2 illustrates examples of the seven perturbation rules.

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

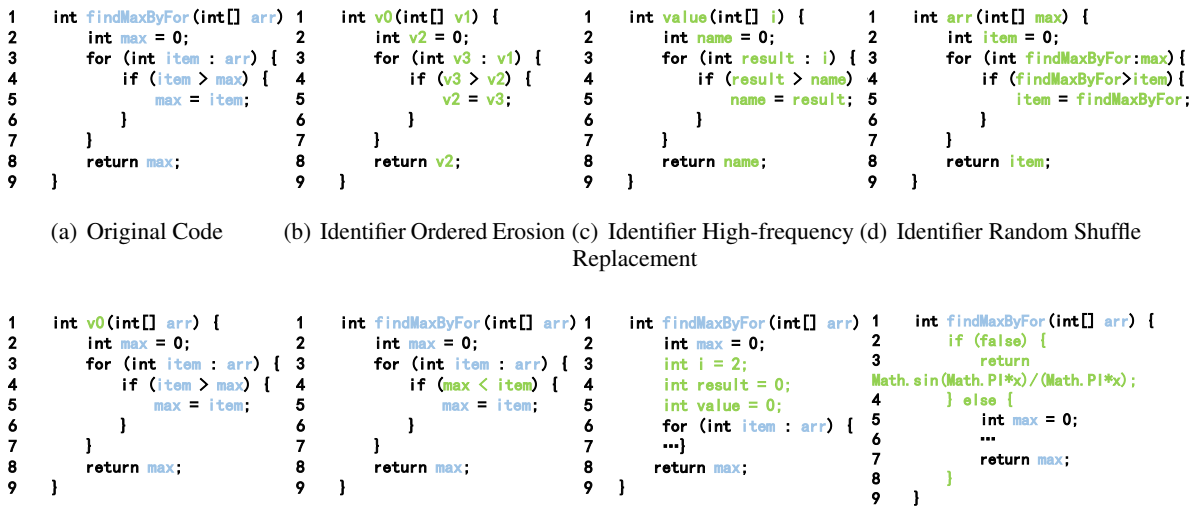
Finally, we apply the seven perturbation rules to the original datasets and extend them into seven obfuscation datasets. We will analyze the fine-grained capability of the model by combining these perturbation rules.

4 Evaluation Methodology

4.1 Baseline Models

We select three code language models as our baseline models: CodeBERT, CodeT5, and CodeLlama. They stand as the state-of-the-art models for code summarization.

CodeBERT (Feng et al., 2020) is a BERT-style pre-training model based on RoBERTa (Liu et al., 2019). The model has been pre-trained on both



(a) Original Code (b) Identifier Ordered Erosion (c) Identifier High-frequency Replacement (d) Identifier Random Shuffle

(e) Function Name Erosion (f) Operators & Operands Swap (g) High-frequency Variable Injection (h) Dead Branch Injection

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

natural and programming languages using two self-supervised objectives, namely, masked language model and replaced token detection. The pre-trained model is finetuned on task-specific datasets for code understanding tasks such as code search and code summarization.

CodeT5 (Wang et al., 2021) is a pre-trained model of code 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 (Roziere et al., 2023) stands as the state-of-the-art LLM for code generation and comprehension. It is built on top of Llama 2(Touvron et al., 2023) 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(Roziere et al., 2023). The infilling method, as illustrated in Figure 4(a), replaces the document content of the input Python code with a special tag, 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. Therefore, there is not an officially recommended inference method in these languages. We generate summaries using the method in bigcode-evaluation-harness[8] which results in reasonable performance. Specifically, we append a prompt at the end of the code to guide the model to output summary text. The prompt template can be found in Figure 4(b) and 4(c) in Appendix.

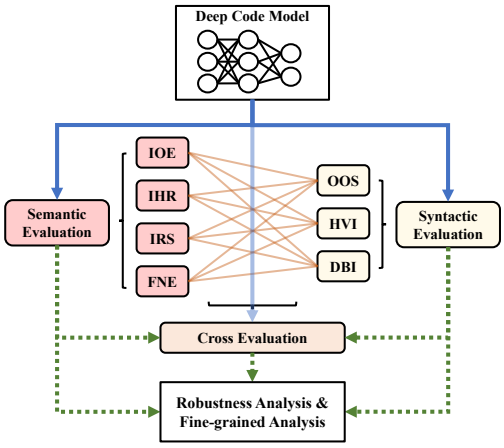


Figure 3: The evaluation workflow of OR-CodeSum, the solid blue arrow indicates the corresponding evaluation of the model and the dashed green arrow indicates the transmission of evaluation results.

4.2 Evaluation WorkFlow

Having obtained the obfuscated datasets, we design an evaluation methodology to comprehensively in-

[6]CodeT5 generation script.

[7]CodeLlama infilling method.

[8]Prompts for code-to-text task with large model.

Dataset	CodeBERT			CodeT5			CodeLlama		
	Python	Go	Java	Python	Go	Java	Python	Go	Java
Primary									
	17.95	17.78	18.62	20.38	19.67	20.66	22.03	12.78	15.11
Semantic Perturb.									
IOE	13.89	**11.02**	13.85	**15.23**	16.90	**14.18**	**11.15**	12.75	13.34
IRS	14.70	13.07	15.42	16.50	17.87	15.88	17.30	12.40	13.95
IHR	**12.54**	12.10	**12.84**	15.84	16.96	14.82	11.86	**11.85**	**12.63**
FNE	14.74	12.95	15.40	17.05	**16.19**	15.72	18.54	12.42	14.17
average	13.97	12.29	14.38	16.16	16.98	15.15	14.71	12.36	13.52
Syntactic Perturb.									
OOS	17.94	17.79	18.61	19.34	19.71	20.68	22.08	12.77	15.08
HVI	17.53	17.75	18.15	19.32	19.62	20.65	21.69	12.88	15.22
DBI	17.34	17.87	18.26	18.77	19.15	20.25	21.69	12.95	14.95
average	17.60	17.80	18.34	19.14	19.49	20.53	21.82	12.87	15.08

Table 3: BLEU scores on the obfuscated datasets.

vestigate the full aspects of model performance. The entire evaluation workflow is summarized in Figure 3. We begin with a robust evaluation that aims to identify the most sensitive aspects of code when the model summarizes source code. This can be achieved by testing the baseline models on the seven obfuscated datasets. For example, when some perturbations have greater impacts on the model, it means that the corresponding aspects (e.g., identifiers) are the most critical to code summarization models. We can also know whether the model is sensitive to semantic or syntactic aspects of the code.

Having identified sensitive aspects of code, we face a new problem: The sensitive aspects of code may obscure the effect of non-sensitive aspects in testing model's ability. For example, if the model heavily relies on semantic features such as identifiers, it will be difficult to identify the impact of syntactic perturbation when the input code contains much semantic information. In order to test models' ability in recognizing low-sensitivity aspects of code, we propose cross-obfuscation datasets that reduce the dominance of high-sensitive aspects of code. Specifically, we perturb the original datasets by combining two perturbation rules, one from semantic perturbation and the other from syntactic perturbation.

Finally, through the two-phase evaluation, we can have a fine-grained analysis of the model. By summing up the results of both robust evaluation and cross-obfuscation evaluation, we can gain more comprehensive insights into the model's ability including performance, capacity, and robustness.

4.3 Implementation Details [9]

We trained all LLMs based on the released pre-trained checkpoint on huggingface[10][11][12]. 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.

4.4 Evaluation Metrics

We used smoothed BLEU-4 score (Lin and Och, 2004) and BERTScore (Zhang et al., 2019) as the evaluation metrics, which are the most widely used metric for shot-text summarization tasks. BLEU is calculated with the same scripts provided by the baseline models. BERTScore computes a sentence-level similarity score by making use of token-level similarities, produced by cosine similarities between their contextual embeddings.

5 Experiments and Analysis [13]

5.1 Robust Evaluation

Table 3 compares the evaluation results of three baseline models on the primary and perturbated datasets, respectively.

[9] Hyperparameters about baseline models can be found in Appendix A

[10] https://huggingface.co/microsoft/codebert-base

[11] https://huggingface.co/Salesforce/codet5-base

[12] https://huggingface.co/codellama/CodeLlama-7b-hf

[13] We show brief results with BLEU scores in the main content. The full results, including the other evaluation metric BERTScore, can be found in Appendix C

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

Table 4: BLEU scores on the cross-obfuscated datasets. The semantic perturbation chooses the one with the greatest impact on the model in each programming language.

Comparing the perturbated datasets Across all perturbations, IOE and IHR has 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 The three baseline models are more sensitive to semantic perturbations than syntax perturbations on code, which means that state-of-the-art code language models pay more attention to semantic features in source code, such as function and variable names.

We particularly notice that the impact of syntactic perturbed datasets is obscured by semantic perturbations. Hence we need a more fine-grained evaluation of the effect of syntactic perturbations.

On the other hand, most downstream tasks of pre-trained code models lack explicit syntactic supervision, which strengthens the binding between the meaningful identifier and the summary hereby hampering the robustness of the model against semantic perturbation.

5.2 Cross-Perturbation Evaluation

Recognizing that models exhibit heightened awareness of semantic code features, we embark on an in-depth examination of their capability in comprehending 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. We construct cross-perturbed 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 [14] and model. These chosen semantic perturbation rules are then paired with one of the syntax perturbation rules.

Table 4 shows the results of the cross-perturbation evaluation. All BLEU scores on the cross-perturbed datasets are compared to datasets that have the most significant impact on models and programming languages.

Comparing the perturbed datasets The results indicate that nearly all baseline models exhibit sensitivity to the perturbation involving *dead branch insertion*. Additionally, the obfuscation technique of *high-frequency variable injection* significantly influences model performance. 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.

[14] More cross-perturbation results in Appendix C

Lang	Model	Semantic	Syntactic	Overall
Python	**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%**

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

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.

5.3 Results Analysis

We provide our comprehensive analysis based on three evaluation aspects: performance, capability, and robustness. In terms of performance (Table 3), CodeLlama outperforms other baseline models on the primary regular test sets, though it shows suboptimal scores when not pretrained on Go and Java. From the capability of the model, all three baseline models tend to focus on the perturbation of semantics over syntax. This is particularly evident in larger language 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.

5.4 Case Study

In addition to the quantitative analysis, we also provide a number of cases in Appendix D.

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 deep code summarization 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 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.

7 Conclusion

In this paper, we propose OR-CodeSum, a robustness evaluation benchmark for code summarization tasks. By extending existing benchmarks with perturbation rules, OR-CodeSum enables a fine-grained 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.

Limitations

The limitations of our work lie in two aspects: 1) The studied baseline models are selected based on

performance. However, our studies indicate that high-performance models do not necessarily have robustness. Studying models with different architectures and training methods may lead to more interesting conclusions. 2) We investigate the robustness of code models through seven obfuscation rules. Other characteristics of code, such as code length, could also affect the performance of the model. For example, 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. We leave the investigation of more code characteristics in future work.

References

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

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*, pages 511–521.

Casey Casalnuovo, E Morgan, and P Devanbu. 2020. Does surprisal predict code comprehension difficulty. In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society*. Cognitive Science Society Toronto, Canada.

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*, pages 18–30.

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):1–28.

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*.

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*, pages 1536–1547, Online. Association for Computational Linguistics.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. *arXiv preprint arXiv:1811.01824*.

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*, pages 1667–1680.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge.

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*.

Paras Jain and Ajay Jain. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.

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*, pages 501–507.

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*.

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*.

Katerina Margatina, Giorgos Vernikos, Loïc Barrault, and Nikolaos Aletras. 2021. Active learning by acquiring contrastive examples. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 650–663.

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)*, pages 165–169. IEEE.

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*.

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

Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022b. 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*, pages 107–119.

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*.

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*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Fan Yan and Ming Li. 2021. Towards generating summaries for lexically confusing code through code erosion. In *IJCAI*, pages 3721–3727.

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

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*.

A Hyperparameters

The hyperparameters and specfications for all language models are provided in Table A.

B Prompt Templates for LLMs

The prompts used for CodeLlama are shown in Figure 4.

	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

Table 6: Hyperparameters of baseline models.

```
def sina_xml_to_url_list(xml_data):
    """<FILL_ME>"""
    rawurl = []
    dom = parseString(xml_data)
    for node in dom.getElementsByTagName('durl'):
        url = node.getElementsByTagName('url')[0]
        rawurl.append(url.childNodes[0].data)
    return rawurl
```

(a) Prompt for Python.

```
func FiltersFromRequest(creq *pb.WatchCreateRequest) []mvcc.FilterFunc {
    filters := make([]mvcc.FilterFunc, 0, len(creq.Filters))
    for _, ft := range creq.Filters {
        switch ft {
        case pb.WatchCreateRequest_NOPUT:
            filters = append(filters, filterNoPut)
        case pb.WatchCreateRequest_NODELETE:
            filters = append(filters, filterNoDelete)
        default:
        }
    }
    return filters
}
Please fill this sentence "The goal of this function is to " in 12 words:
```

(b) Prompt for Go.

```
public long calculateDelay(TimeUnit unit) {
    float delta = variancePercent / 100f; // e.g., 20 / 100f == 0.2f
    float lowerBound = 1f - delta; // 0.2f --> 0.8f
    float upperBound = 1f + delta; // 0.2f --> 1.2f
    float bound = upperBound - lowerBound; // 1.2f - 0.8f == 0.4f
    float delayPercent = lowerBound + (random.nextFloat() * bound); // 0.8 +
(rnd * 0.4)
    long callDelayMs = (long) (delayMs * delayPercent);
    return MILLISECONDS.convert(callDelayMs, unit);
}
Please fill this sentence "The goal of this function is to " in 12 words:
```

(c) Prompt for Java.

Figure 4: Prompts used for CodeLlama inference.

Dataset	CodeBERT			CodeT5			CodeLlama		
	Python	**Go**	**Java**	**Python**	**Go**	**Java**	**Python**	**Go**	**Java**
Primary									
	17.95	17.78	18.62	20.38	19.67	20.66	22.03	12.78	15.11
Semantic Perturb.									
IOE	13.89	11.02	13.85	15.23	16.90	14.18	11.15	12.75	13.34
IRS	14.70	13.07	15.42	16.50	17.87	15.88	17.30	12.40	13.95
IHR	12.54	12.10	12.84	15.84	16.96	14.82	11.86	11.85	12.63
FNE	14.74	12.95	15.40	17.05	16.19	15.72	18.54	12.42	14.17
Syntactic Perturb.									
OOS	17.94	17.79	18.61	19.34	19.71	20.68	22.08	12.77	15.08
HVI	17.53	17.75	18.15	19.32	19.62	20.65	21.69	12.88	15.22
DBI	17.34	17.87	18.26	18.77	19.15	20.25	21.69	12.95	14.95
Cross Perturb.									
IOE × OOS	14.07	10.93	13.87	14.21	16.90	14.34	11.07	12.69	13.31
IOE × HVI	13.21	10.81	13.26	13.75	16.66	13.58	8.85	12.74	13.27
IOE × DBI	12.58	10.74	13.20	12.70	16.31	13.31	8.15	12.43	12.48
IRS × OOS	14.70	12.88	15.38	15.54	17.90	15.94	16.96	12.43	14.00
IRS × HVI	14.05	12.17	14.67	15.32	17.73	15.71	15.29	12.61	14.08
IRS × DBI	13.16	12.13	14.71	13.93	16.99	14.95	14.94	12.20	13.56
IHR × OOS	12.52	12.00	12.87	14.45	16.91	14.82	11.89	11.64	12.50
IHR × HVI	11.57	11.57	12.16	14.16	16.72	14.61	9.67	11.82	12.51
IHR × DBI	11.49	11.25	12.13	13.23	16.08	13.78	8.66	11.41	12.07
FNE × OOS	14.69	12.95	15.37	15.97	16.25	15.89	18.41	12.42	14.18
FNE × HVI	13.66	12.60	14.46	15.22	15.75	14.58	16.29	12.46	14.21
FNE × DBI	12.94	12.16	14.27	14.25	14.58	15.11	15.73	12.09	13.28

Table 7: **Smoothed BLEU-4** evaluation results.

Dataset	CodeBERT			CodeT5			CodeLlama		
	Python	**Go**	**Java**	**Python**	**Go**	**Java**	**Python**	**Go**	**Java**
Primary									
	29.64	40.11	31.92	34.41	43.18	35.35	38.91	26.30	33.68
Semantic Perturb.									
IOE	19.52	16.26	21.53	22.05	37.09	20.90	13.27	26.26	29.40
IRS	22.82	23.28	25.50	26.97	37.64	26.51	30.03	25.54	31.43
IHR	17.53	22.38	19.61	25.03	36.26	24.65	20.02	24.01	27.85
FNE	22.63	21.48	25.17	26.99	35.73	22.60	31.78	25.18	31.28
Syntactic Perturb.									
OOS	29.63	40.14	31.90	33.43	43.27	35.45	38.86	26.29	33.70
HVI	28.69	40.08	30.99	33.19	43.00	35.25	38.53	26.46	33.67
DBI	28.27	40.40	31.41	31.61	42.51	34.87	38.49	26.28	33.06
Cross Perturb.									
IOE $\times$ OOS	19.71	16.05	21.56	20.95	37.08	21.42	13.19	26.24	29.32
IOE $\times$ HVI	16.57	15.96	20.22	19.11	36.85	18.79	6.76	26.26	29.18
IOE $\times$ DBI	16.71	16.03	20.05	16.82	36.51	20.06	6.43	25.70	27.51
IRS $\times$ OOS	22.79	22.69	25.48	25.98	37.68	26.64	29.25	25.37	31.51
IRS $\times$ HVI	21.16	21.76	23.90	25.15	37.44	25.94	25.26	25.57	30.98
IRS $\times$ DBI	18.92	22.79	24.38	21.09	36.59	24.74	24.43	24.89	30.20
IHR $\times$ OOS	17.52	22.36	19.69	23.42	36.88	24.85	19.62	23.85	27.81
IHR $\times$ HVI	15.29	21.49	17.99	22.49	36.63	23.82	14.44	23.93	27.65
IHR $\times$ DBI	14.89	21.63	18.03	18.99	35.81	21.97	11.50	22.87	26.34
FNE $\times$ OOS	22.52	21.52	25.10	25.89	35.83	23.15	31.47	25.23	31.22
FNE $\times$ HVI	19.83	20.95	23.10	22.50	33.98	18.01	26.11	25.35	31.19
FNE $\times$ DBI	18.15	21.33	22.71	20.91	32.64	22.65	24.88	24.62	29.67

Table 8: **BERTScore** evaluation results. All values are multiplied by 100 for ease of display

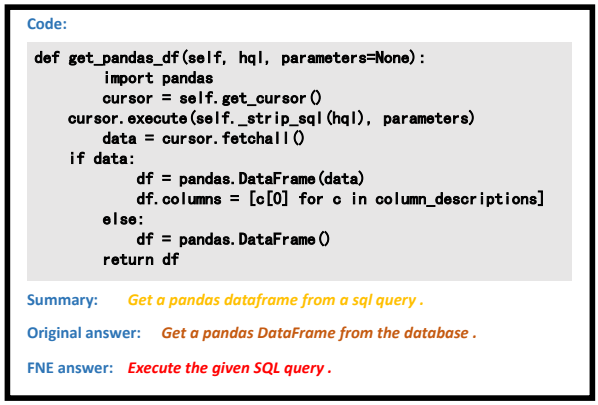
C Full Results and Results with Other Metrics

Table 7 and 8 show the full results using BLEU-4 and BERTScore respectively. BERTScores often lie in a small range. Following the general practice, we rescale them with baselines to fall in [0, 1][15].

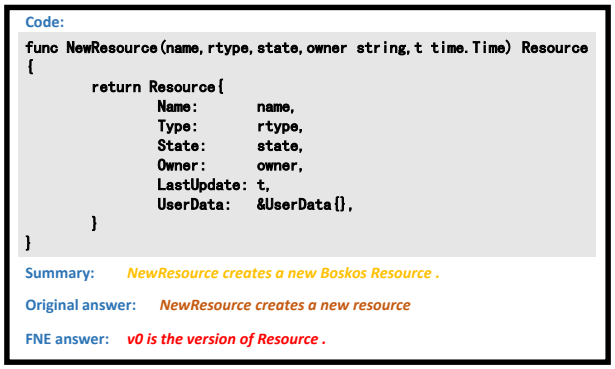
The findings from BERTScore align closely with those derived from BLEU. Generally, as the score increases, the model tends to exhibit greater vulnerability to semantic perturbations. Notably, among all semantic perturbations, IOE and IHR exhibit the most significant impact. Among syntax perturbations, DBI has the most impact on model performance.

D Case Study

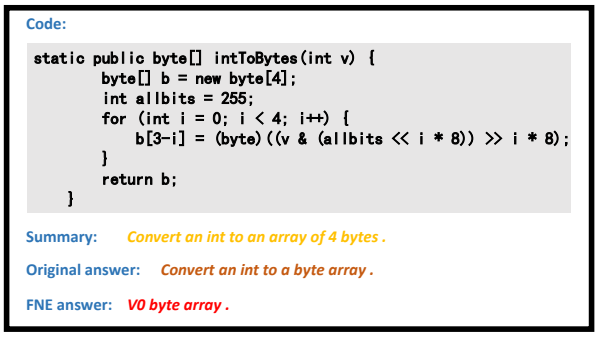
Figure 5 presents three cases of code summarization by CodeT5. We compare the original human-written summary to model-generated summaries with and without FNE perturbation, respectively. We find that when the function name is eroded, the summary generated by CodeT5 becomes confusing. The example suggests that CodeT5 focuses more on semantic cues such as function names.



(a) A case in Python.



(b) A case in Go.



(c) A case in Java.

Figure 5: Summaries generated by CodeT5 with and without FNE perturbation.

[15] Rescaling BERTScore with Baselines