Uncovering LLM-Generated Code: A Zero-Shot Synthetic Code Detector via Code Rewriting

Tong Ye[1*], **Yangkai Du**[1*], **Tengfei Ma**[2], **Lingfei Wu**[3],
Xuhong Zhang[1†], **Shouling Ji**[1], **Wenhai Wang**[1]
[1]Zhejiang University
[2]Stony Brook University
[3]Anytime.AI
{tongye,yangkaidu,zhangxuhong,sji,zdzzlab}@zju.edu.cn, tengfei.ma@stonybrook.edu, lwu@anytime-ai.com

Abstract

Large Language Models (LLMs) have demonstrated remarkable proficiency in generating code. However, the misuse of LLM-generated (synthetic) code has raised concerns in both educational and industrial contexts, underscoring the urgent need for synthetic code detectors. Existing methods for detecting synthetic content are primarily designed for general text and struggle with code due to the unique grammatical structure of programming languages and the presence of numerous "low-entropy" tokens. Building on this, our work proposes a novel zero-shot synthetic code detector based on the similarity between the original code and its LLM-rewritten variants. Our method is based on the observation that differences between LLM-rewritten and original code tend to be smaller when the original code is synthetic. We utilize self-supervised contrastive learning to train a code similarity model and evaluate our approach on two synthetic code detection benchmarks. Our results demonstrate a significant improvement over existing SOTA synthetic content detectors, delivering notable gains in both performance and robustness on the APPS and MBPP benchmarks.

Introduction

LLMs and Code LLMs have shown remarkable capability in understanding and generating code (Chen et al. 2021; Fried et al. 2022; Nijkamp et al. 2022; Rozière et al. 2023; Guo et al. 2024). Those LLMs can function as professional coding assistants for programmers, offering intelligent code completion and document generation capabilities, such as Github Copilot (Github 2021).

The breakthrough of LLMs has greatly improved coding efficiency and lowered the barrier to programming (Kazemitabaar et al. 2023). However, this also raises concerns about misuse. In education, students are using LLMs to complete coding assignments and exams, making it harder to assess their true abilities (Kazemitabaar et al. 2023; Denny et al. 2024). A study shows GPT-4 can solve LeetCode problems at an average human level (Bubeck et al. 2023), increasing the risk of cheating. Moreover, LLM-generated code often contains security vulnerabilities, posing risks in industrial applications (He and Vechev 2023). An evaluation found 40% of Copilot-generated programs contain dangerous vulnerabilities (Pearce et al. 2022), highlighting the need for a more rigorous review of synthetic code.

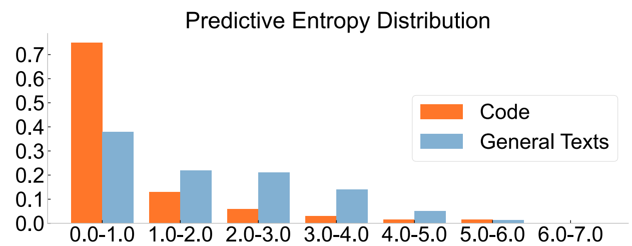

Figure 1: Token Entropy Distribution in General Texts vs. Code, Estimated by Llama-13B (Touvron et al. 2023a).

Building on these critical real-world concerns, developing a synthetic code detector is essential to address the misuse of LLM-generated code. While efforts have been made to detect LLM-generated text (Gehrmann, Strobelt, and Rush 2019; Mitchell et al. 2023; Zellers et al. 2019; Ippolito et al. 2020; Zhong et al. 2020; Pu et al. 2023), none specifically target code. The effectiveness of these methods on code content remains uncertain. Unfortunately, our experiments show that state-of-the-art detectors like GLTR (Gehrmann, Strobelt, and Rush 2019) and DetectGPT (Mitchell et al. 2023) suffer a significant performance drop (around 32% in AUROC) when applied to code. We thoroughly analyze the root causes of existing methods' failures, as shown below.

Challenges in Applying Existing Detection Methods to Code. Our observations and meticulous analysis reveal that the core issue lies in the fundamental logic of existing state-of-the-art text detection methods, which rely on the statistical log probability of tokens. LLMs tend to assign higher log probabilities to the tokens they generate due to their strong confidence in these self-generated tokens. Detection methods capitalize on this trait to distinguish between model-generated and human-written text, achieving high accuracy. However, in the domain of code, this approach exposes its inherent weaknesses. Unlike natural

*These authors contributed equally.
†Corresponding Author.

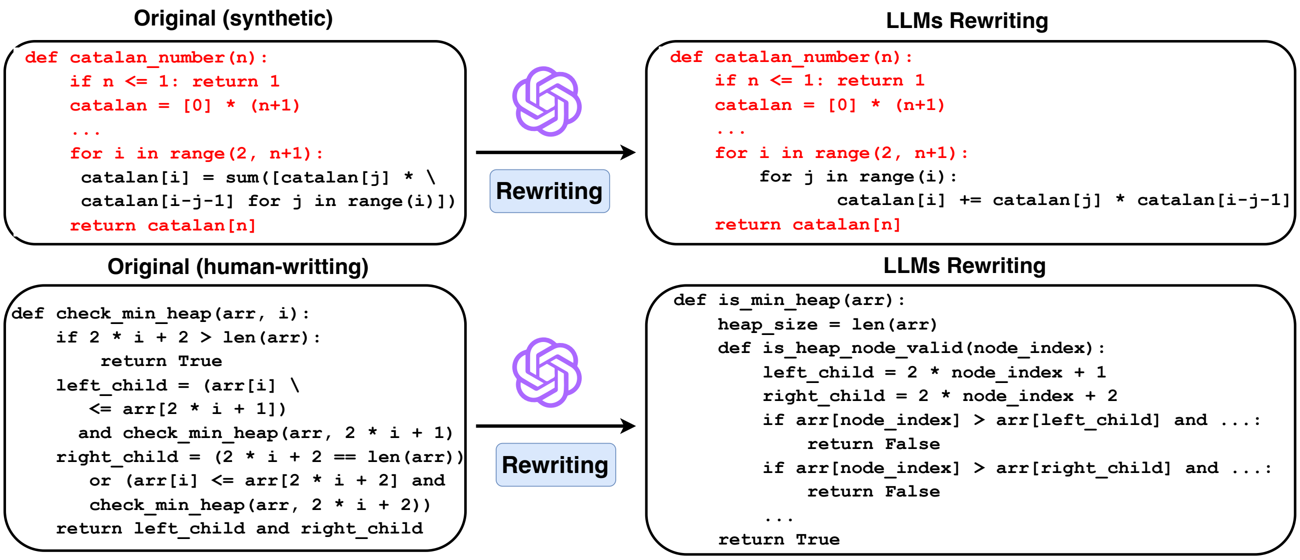

Figure 2: Rewriting Original Code (synthetic & human-writting) with GPT-3.5-Turbo.

language, code typically follows a more rigid grammatical structure. In specific programming languages and contexts, the next code token is often deterministic or has very limited options. For example, in Python, functions always start with "def", followed by a ":" after the function signature "def function_name()". These token are what we refer to as "low-entropy" tokens. The prevalence of such "low-entropy" tokens in code greatly reduces the effective tokens available for text detection methods to distinguish between human-written and LLM-generated code, leading to statistically insignificant differences between the two. Our investigation into the entropy distribution of code tokens versus text tokens (Figure 1) confirms that the entropy distribution for text tokens is more varied compared to code, with over 70% of code tokens having an entropy of less than 1.

Observations and Hypothesis. Therefore, to effectively detect synthetic code, it is crucial to move beyond the current focus on token log probabilities and adopt a more holistic, global perspective. Based on this insight, we observed that when LLMs rewrite synthetic code—whether generated by themselves or other LLMs—they tend to produce output that closely mirrors the original code. In contrast, when rewriting human-written code, the LLM-generated output diverges more significantly. As illustrated in Figure 2, the similarity between synthetic code and its rewritten version is striking (marked in red), while human-written code and its rewritten counterpart show more substantial differences.

This observation led to our hypothesis that the similarity between the original and rewritten code can serve as an indicator for detecting synthetic code. By focusing on code rewriting and similarity measurement, we can avoid reliance on statistical token likelihood estimation, thereby addressing the root cause of failures in existing text detectors for code. To validate this hypothesis, we propose a zero-shot synthetic code detector using Code Rewriting and Similarity Measurement. Our method consists of three crucial steps: First, we use an LLM with an appropriate prompt to rewrite a given piece of code, producing a pair of the original and rewritten code. Next, to accurately gauge the similarity between the original and rewritten code, we then develop a code similarity model that predicts a similarity score for the given code pair. Finally, using the cosine similarity metric, we estimate the expected similarity score by sampling m rewritten codes following the identical procedure and averaging their scores.

During our experiments, due to the lack of existing datasets for synthetic code detection, we constructed two new datasets based on APPS (Hendrycks et al. 2021) and MBPP (Austin et al. 2021). To validate the universal applicability of our hypothesis and method across different code generation tools, we generated synthetic code using two open-source LLMs (CodeLlama and StarChat) and two proprietary APIs (GPT-3.5-Turbo and GPT-4). Our experimental results demonstrate that our proposed method significantly outperforms current state-of-the-art approaches, achieving a 20.5% improvement in AUROC on the APPS benchmark and a 29.1% improvement on the MBPP benchmark. Additionally, our method is highly practical and requires minimal resources and permissions, only needing the ability to perform LLMs inference or access their APIs. This contrasts with previous detection methods that rely on knowledge of token log probabilities.

Overall, the main contributions are three folds:

- We identify a significant performance gap when applying synthetic content detectors designed for general text to the code domain and provide a detailed analysis to uncover the reasons for this discrepancy.
- We propose a novel zero-shot synthetic code detection approach that leverages code rewriting and similarity measurement. This method is applicable to both open-source Code LLMs and closed-source LLMs.

- Extensive experiments demonstrate the effectiveness of our method, showing significant improvements in both accuracy and robustness compared to existing methods.

Related Works

LLMs for Code Generation

With the rapid rise in popularity of LLMs, a growing number of studies have focused on Code LLMs aimed at automating software engineering (Chen et al. 2021; Guo et al. 2024). Code generation is a crucial capability of these models, as demonstrated by pioneering works like Codex (Chen et al. 2021), AlphaCode (Li et al. 2022b), CodeGeeX (Zheng et al. 2023), CodeLlama (Rozière et al. 2023), and GPT-4 (Achiam et al. 2024). As LLMs continue to advance, professionals across various industries are increasingly integrating them into their daily workflows. According to a community survey by StackOverflow (2023), 44% of experienced developers and 55% of beginners are already using AI coding assistants, with the majority relying on two main tools: ChatGPT (83%) and GitHub Copilot (56%). Consequently, the automated detection of whether code is human-written or LLM-generated has become increasingly important.

Detection of Synthetic Text

Detecting AI-generated (synthetic) text has been a focus of research even before the emergence of LLMs. The primary approaches to synthetic text detection fall into two categories. The first treats detection as a binary classification problem, using synthetic texts from generative models to train supervised models, often based on neural networks, such as Transformers (Zellers et al. 2019; Ippolito et al. 2020; Zhong et al. 2020; Bakhtin et al. 2019; Uchendu et al. 2020). The second approach involves designing zero-shot metrics (Gehrmann, Strobelt, and Rush 2019; Su et al. 2023; Mitchell et al. 2023) to measure the relationship between a given text and the distribution of generative models. These methods all rely on the premise that LLMs generate tokens with higher confidence, reflected in elevated log probabilities. By statistically analyzing log probabilities, effective differentiation can be achieved. Contemporaneous study Mao et al. (2024) fails to reveal the "low-entropy" nature specific to code. All of the above underscores the need for a code-specific detection method from a new perspective.

Methodology

Problem Definition. We focus on the zero-shot synthetic code detection problem, where the task is to determine whether a code snippet x is generated by an LLM or written by a human. In this context, "zero-shot" implies that we do not require a labeled dataset of synthetic and human-written code for training. Unlike previous zero-shot approaches that assume a "white-box" setting with access to the generative model's log probability scores, we argue that this assumption is too strong for practical synthetic code detection. Many commercial LLMs provide only the generated content, without revealing the underlying log probability scores. Therefore, we investigate the more stricter "black-box" setting, where only the generated content is available.

Algorithm 1: Zero-shot Synthetic Code Detection

Input: x: code snippet, $\mathcal{G}$: generative model, $\mathcal{M}$: similarity model, m: number of rewriting, ϵ: threshold.
Output: `true`: x is generated by $\mathcal{G}$, `false`: x is not generated by $\mathcal{G}$.

1: // generate m rewriting of x.
2: $x'_i \sim \mathcal{G}(\cdot \mid x),\ i \in [1...m]$
3: // estimate the similarity score.
4: $score \leftarrow \frac{1}{m}\sum_i \frac{\mathcal{M}(x)\cdot\mathcal{M}(x'_i)^T}{\|\mathcal{M}(x)\|_2 \times \|\mathcal{M}(x'_i)\|_2}$.
5: **if** $score > \epsilon$ **then**
6: return `true` // x is generated by $\mathcal{G}$.
7: **else**
8: return `false` // x is not generated by $\mathcal{G}$.
9: **end if**

Intuition and Hypothesis. Distinctively, we adopt a holistic and global perspective, grounded in the intuition that every programmer develops a unique coding style, shaped by their routines and habits. Similarly, generative models, like experienced programmers, exhibit distinctive coding patterns influenced by the biases present in their training data. Therefore, LLMs can be viewed as programmers with consistent coding styles shaped by these inherent biases.

Therefore, based on consistent code writing patterns, we hypothesize that when generative LLMs are tasked with rewriting synthetic code, the differences between the rewritten and original code will be smaller compared to human-written code. Figure 2 visually illustrates this concept. Building on this hypothesis, we propose a zero-shot synthetic code detection method that leverages Code Rewriting and Similarity Measurement. The design principle is straightforward: for a given code snippet x, we perform multiple code rewriting using a generative LLM $\mathcal{G}$. Then, we measure the average similarity between the rewritten and original code using a code similarity model $\mathcal{M}$, trained with self-supervised contrastive learning (Gao, Yao, and Chen 2021) on unlabeled code. This average similarity score serves as the detection metric. Our approach is outlined in Algorithm 1, with detailed explanations of Code Rewriting and Similarity Measurement in the following sections.

```
### Code:
{src_code}

### Instruction:
Please explain the functionality of
↪ the given code, then rewrite it in
↪ a single markdown code block. No
↪ additional clarifications.
```

Figure 3: Prompt for Code Rewriting.

Code Rewriting. Given a code snippet x, we generate a rewriting of x utilizing the chain of thought prompting method (Wei et al. 2022). We prompt the LLM $\mathcal{G}$ with the original x, instructing it to first generate an explanation of the code, followed by a potential rewrite x', as shown in Figure 3. The intermediate code explanation can help LLM understand the original code and generate a valid rewrite according to the explanation. To minimize noise and prevent evasion tactics, we normalize x by removing comments and empty lines before prompting. The LLM is instructed to return the rewritten code in Markdown format, and we remove any in-line comments during post-processing.

Similarity Measurement. To accurately measure the similarity between the original code and the rewritten code, we require a code similarity model $\mathcal{M}$ that can predict a similarity score S for a given code pair. Code similarity learning is a pivotal research area in AI for software engineering, often centered on code representation learning. This involves creating dense semantic representations using various code structures (Feng et al. 2020; Ye et al. 2021; Wang et al. 2021; Guo et al. 2021; Wang et al. 2022; Li et al. 2022a) and measuring similarity through vector distances. We selected GraphCodeBERT (Guo et al. 2021) as our base model due to its widespread adoption, availability, and extensive pre-training on large-scale code data.

To enhance the code function-level representation of GraphCodeBERT, we employ self-supervised contrastive learning. Following SimCSE (Gao, Yao, and Chen 2021), we adapt the unsupervised SimCSE method to the code domain, using standard dropout as the data augmentation method for contrastive learning. For a given code snippet x, we take the last-layer hidden states of [CLS] token as code representation. During the unsupervised training stage, we introduce an MLP layer to obtain the final representation $\mathbf{h}$ of x.

Formally, for a batch of input code snippets $\{x_i\}_{i=1}^N$, we pass the batch through GraphCodeBERT twice, obtaining two sets of embeddings, $\{\mathbf{h}_i\}_{i=1}^N$ and $\{\mathbf{h}'_i\}_{i=1}^N$, each with a different dropout mask applied. The training objective of SimCSE is defined as:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} log \frac{e^{sim(\mathbf{h}_i,\mathbf{h}'_i)/\tau}}{\sum_{j=1}^{N} e^{sim(\mathbf{h}_i,\mathbf{h}'_j)/\tau}} \tag{1}$$

Here, τ is a temperature hyperparameter set to 0.1. The term $sim(\mathbf{h}_i, \mathbf{h}'_i)$ is the cosine similarity $\frac{\mathbf{h}_i \cdot \mathbf{h}'_i}{|\mathbf{h}_i||\mathbf{h}'_i|}$.

After the unsupervised contrastive learning stage, the MLP layer is removed, and we use only the last-layer hidden states of the [CLS] token as the code snippet representation, following (Gao, Yao, and Chen 2021). For a given code snippet x and its rewriting x' generated by $\mathcal{G}$, we obtain the final representations of x and x' by feeding them into the similarity model $\mathcal{M}$. It is important to note that our framework is not tied to a specific similarity model; $\mathcal{M}$ can be any implementation that effectively models code similarity, such as OpenAI's text embedding services (Greene et al. 2022).

Finally, we employ the cosine similarity function and estimate the expectation of the similarity score by sampling m rewritten code, formally expressed as:

$$score(x) = \mathrm{E}_{x' \sim \mathcal{G}(\cdot|x)} sim(\mathcal{M}(x), \mathcal{M}(x')) \tag{2}$$

While a larger m can provide a more accurate estimation, it also requires generating more rewrites. Excitingly, our experiments show that using just 4 rewrites is sufficient to achieve excellent detection performance.

Experiment Setting

Benchmarks. Due to the lack of existing benchmarks for evaluating synthetic code detectors, we developed two Python-based benchmarks using APPS (Hendrycks et al. 2021) and MBPP (Austin et al. 2021). To generate the synthetic code, we used the natural language descriptions from these datasets as prompts and employed four widely-used code generation tools. For these four generation tools, we selected two popular open-source Code LLMs, CodeLlama-13B-Instruct (Rozière et al. 2023) and StarChat-Alpha (Tunstall et al. 2023), as well as two leading proprietary generation APIs, GPT-3.5-Turbo (OpenAI 2022) and GPT-4 (Achiam et al. 2024). Details are provided in Arxiv version.

Metrics. We evaluate all detectors on our benchmarks using the Area Under the Receiver Operating Characteristic (AUROC) curve, following Mitchell et al. (2023).

Baselines. To demonstrate the effectiveness of our zero-shot synthetic code detector, we adapt previous zero-shot AI content detectors for general text to code content. Specifically, we consider the following zero-shot detection methods that use a surrogate model to approximate the true distribution of the generative model: **log P(x)** (Gehrmann, Strobelt, and Rush 2019), **LogRank**, **Rank**, **Entropy** (Mitchell et al. 2023), **LRR**, **NPR** (Su et al. 2023), and **DetectGPT** (Mitchell et al. 2023). To effectively adapt these methods for code domain, we replaced the surrogate score models with LLMs specifically trained on large-scale code datasets, ensuring a fair comparison. Additionally, we compared two supervised detectors against the zero-shot methods: **GPTZero** (GPTZero 2023), a leading AI content detection service trained on millions of synthetic texts from various models, including GPT-4 and Bard, and **OpenAI-Detector** (OpenAI 2019), an open-source detector based on RoBERTa-large (Liu et al. 2019) and trained on GPT-2 outputs. Detailed descriptions of above baselines are shown in Arxiv version.

Detector (Rewriting) LLMs. Our approach necessitates a rewriting LLM, $\mathcal{G}$, to serve as the detector LLM. We evaluate CodeLlama, StarChat-Alpha, and GPT-3.5-Turbo as detector LLMs for our method while using CodeLlama and StarChat-Alpha for baseline comparison. For code rewriting, we utilize nucleus sampling with a top-p of 0.95 and a temperature of 0.8.

Similarity Model Training. We continue to train a code similarity model using unsupervised SimCSE, starting from the initial GraphCodeBERT. For the SimCSE training, we collect thousands of code snippets from publicly available code-related datasets as our training data. Once training is complete, the GraphCodeBERT model is fixed for all subsequent experiments. The training data, training details and hyperparameters are provided in Arxiv version.

Experiment Results

Main Results. We list the AUROC score of all zero-shot detectors on the APPS and MBPP benchmarks in Table

Dataset	APPS					MBPP				
Generators	CodeLlama	StarChat	GPT-3.5	GPT-4	**Avg.**	CodeLlama	StarChat	GPT-3.5	GPT-4	**Avg.**
GPTZero	52.71	56.25	53.68	58.24	55.22	59.53	60.82	57.29	61.55	59.80
OpenAI	56.32	50.08	48.48	55.81	52.67	48.81	47.40	43.31	46.44	46.49
Using CodeLlama as Detector LLM										
log P(x)	66.14	59.40	64.58	59.27	62.35	50.70	53.84	63.05	53.35	55.24
LogRank	69.79	61.54	67.31	62.89	65.38	60.76	58.56	68.05	58.91	61.57
Rank	52.17	48.63	50.77	48.04	49.90	25.99	35.75	42.03	36.33	35.03
Entropy	58.91	54.71	61.49	55.87	57.75	37.22	44.90	50.88	43.40	44.10
DetectGPT	61.28	57.71	62.06	53.41	59.85	56.28	53.18	66.56	63.84	59.96
LRR	67.15	62.16	67.82	60.06	64.30	53.25	56.32	64.29	54.62	57.12
NPR	65.49	60.08	66.53	58.62	62.68	54.37	55.10	68.85	64.96	60.82
Ours $m = 2$	80.78	72.91	73.12	68.19	73.75	77.90	68.88	76.36	75.02	74.54
Ours $m = 4$	85.42	76.53	77.70	74.29	78.49	82.91	71.50	79.83	77.71	77.99
Ours $m = 8$	**87.77**	78.13	80.23	74.51	80.16	**86.21**	75.70	83.58	81.75	**81.81**
Using StarChat as Detector LLM										
log P(x)	66.41	65.27	65.54	62.18	64.85	55.81	64.86	69.91	60.17	62.69
LogRank	66.95	65.81	66.74	64.25	65.93	58.69	65.31	69.56	59.55	63.28
Rank	53.85	48.24	50.37	49.77	50.56	37.24	44.48	47.97	46.47	44.04
Entropy	56.55	55.43	59.60	55.30	56.72	39.03	48.72	55.35	47.22	47.33
DetectGPT	60.92	58.23	61.52	58.62	61.26	54.41	55.74	66.49	65.02	60.42
LRR	66.55	68.91	68.45	65.88	67.45	56.80	66.74	69.87	60.16	63.39
NPR	64.47	63.60	66.43	65.00	64.88	54.20	60.00	70.43	66.17	62.70
Ours $m = 2$	81.93	77.23	72.46	72.89	76.13	79.68	73.79	79.28	69.24	75.50
Ours $m = 4$	85.51	79.24	74.58	77.35	79.17	80.61	76.44	81.05	74.67	78.19
Ours $m = 8$	87.24	81.35	76.28	77.84	80.68	83.67	79.00	83.17	78.04	80.97
Using GPT-3.5-Turbo as Detector LLM										
Ours $m = 2$	77.84	81.67	79.02	79.04	79.39	66.21	77.29	83.05	83.45	77.50
Ours $m = 4$	78.21	82.22	82.12	78.69	80.31	67.00	78.87	85.39	82.23	78.37
Ours $m = 8$	78.47	**82.48**	**83.25**	**80.87**	**81.27**	67.66	**79.23**	**86.23**	**84.00**	79.28

Table 1: Main Results. The first two rows list the two benchmarks and their corresponding four generation tools. The subsequent sections detail the AUROC scores of our methods, compared to seven other zero-shot detectors, using three Detector LLMs.

1. Among the seven baselines, LogRank, LRR, and NPR are the most effective methods. However, the performance of baseline zero-shot detectors, including log P(x), DetectGPT, LRR, and NPR, drops significantly on code benchmarks compared to their reported performance on general text detection (Mitchell et al. 2023; Su et al. 2023). This decline is consistent across different generation tools and detector LLMs. The primary reason, as mentioned earlier, is that SOTA zero-shot text detectors rely on token log probabilities, which are less effective for code due to their uniform grammatical structure and massive "low entropy" tokens. Additionally, we observe that baseline performance improves when the Detector LLM and generation tool are identical, but declines when they differ.

As shown in Table 1, our proposed methods significantly outperform previous approaches across all four code generation tools, with a 20.5% improvement on the APPS benchmark and a 29.1% improvement on the MBPP benchmark. This success is largely due to our holistic viewpoint to detecting synthetic code, which moves beyond token-wise scoring. By employing code rewriting and similarity measurement, our method effectively addresses the limitations of previous techniques in the code domain, leveraging the consistent coding style of LLMs and their tendency to generate the most confident, likely code. Notably, our method achieves strong detection performance with just two rewritings for estimating similarity. Increasing the number of rewritings ($m = 4$ or $m = 8$) further enhances performance.

Effectiveness of Detector LLM. We observe that GPT-3.5-Turbo performs optimally as the detector LLM for identifying code generated by StarChat, GPT-3.5-Turbo, and GPT-4, but not for CodeLlama. The best detection results for CodeLlama-generated code are achieved when using CodeLlama itself as the detector. This is likely because StarChat is fine-tuned on an instruction dataset distilled from GPT-3.5-Turbo and GPT-4, leading to a closer distribution among these three models. In contrast, CodeLlama, trained on a dataset derived from self-instructing Llama-2 (Touvron et al. 2023b), exhibits a different distribution, explaining GPT-3.5-Turbo's suboptimal performance in detecting CodeLlama-generated code. Regarding the superior performance of GPT-3.5-Turbo over StarChat in detecting StarChat-generated code, we found that StarChat tends to oversimplify rewrites for complex problems, whereas GPT-3.5-Turbo provides more accurate and complete rewrites.

Comparison to Supervised Detector. The first two rows in Table 1 display the results of the supervised detectors, GPTZero and OpenAI-Detector. Despite being trained on millions of labeled samples, GPTZero and OpenAI-Detector perform no better than random guessing when applied to code content. This suggests that these supervised detectors may overfit to their training distribution and struggle to generalize to new domains without adaptive tuning. In contrast, our proposed zero-shot detection methods show strong generalization across different code distributions and outperform the supervised detectors.

Ablation Study. We conducted an ablation experiment to assess the contributions of the two primary components in our design: Code Rewriting and Similarity Measurement. We considered two ablation settings: First, replacing the Code Rewriting with in-fill perturbation following DetectGPT (Mitchell et al. 2023) while retaining the Similarity Measurement (Sim); Second, retaining Code Rewriting (CR) and replacing the Similarity Measurement with token-wise score difference (LogProb, LogRank, Rank and Entropy). The results are presented in Table 2.

Dataset	APPS			
Generators	CodeLlama	StarChat	GPT-3.5	GPT-4
CR + LogProb	57.29	46.70	50.58	55.13
CR + LogRank	75.28	64.76	75.85	67.69
CR + Rank	62.34	55.42	60.59	59.34
CR + Entropy	60.39	59.42	68.20	59.35
Perturb + Sim	74.55	71.55	74.23	53.43
CR + Sim	**87.77**	**76.96**	**80.23**	**74.51**

Table 2: Ablation Study. All results are reported when using CodeLlama as the Detector LLM.

The ablation results indicate that our "CR + Sim" method significantly enhances the detection performance. Replacing either Code Rewriting or Similarity Measurement leads to a noticeable decrease in performance, underscoring the importance of these two components.

Deeper Analysis

To further explore the effectiveness and robustness of our methods, we conducted comprehensive experiments to evaluate performance across various settings and scenarios. All experiments in this section are conducted using GPT-3.5-Turbo as the generation tool.

Choice of Similarity Model. In the Methodology section, we exploit self-supervised contrastive learning to train a better code similarity model based on GraphCodeBERT (GCB-SimCSE). To investigate the impact of the Similarity Model, we experiment with three other variants of code similarity models: UniXcoder (Guo et al. 2022) trained by SimCSE (Unix-SimCSE), the original GraphCodeBERT model with average pooling (GCB-avg) and OpenAI's text embedding service (Text-ada-002) (Greene et al. 2022). The results on the APPS benchmark are listed in Table 3. While Unix-SimCSE achieves the highest performance, the other three models still outperform previous zero-shot detectors.

Detectors	CodeLlama	StarChat	GPT-3.5
Unix-SimCSE	**81.21**	**77.84**	**86.37**
GCB-SimCSE	80.23	76.28	83.25
GCB-avg	72.79	70.95	72.22
Text-ada-002	70.18	69.29	76.48

Table 3: Choice of different similarity models. We use CodeLlama, StarChat and GPT-3.5-turbo as Detector LLMs.

These findings suggest that our method is not reliant on a single similarity model. Additionally, since UniXcoder is an enhanced version of GraphCodeBERT, the superior performance of Unix-SimCSE over our original GCB-SimCSE indicates that detection performance can be further improved by using a stronger and more robust similarity model.

Impact of Decoding Strategy. Adjusting the temperature parameter in LLMs balances output diversity and accuracy, with lower temperatures yielding more deterministic and consistent results, while higher temperatures produce more varied and creative outputs. To investigate this, we conducted experiments on the APPS and MBPP benchmarks, varying the generator temperature from $[0.2, 0.4, 0.8]$ while keeping the rewriting temperature fixed at 0.8. This range was chosen based on the Codex (Chen et al. 2021), which found $T = 0.2$ and $T = 0.8$ optimal for pass@1 and pass@100 rates, respectively. The results in Figure 4 demonstrate that our method exhibits superior consistency across different temperatures.

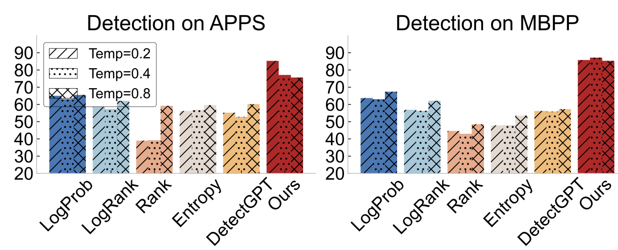

Figure 4: Impact of Decoding Strategy.

Generalizability to Different Programming Languages. To assess the generalizability of our method to different programming languages, we construct an additional C++ benchmark using the Code Contest dataset (Li et al. 2022b). We generate synthetic code using the same four-generation tools. The detection performance on C++ is presented in Table 4. Our method still achieves notable improvement compared to other zero-shot baselines on C++ benchmarks. Moreover, we observe that synthetic C++ code is easier to detect for both our method and other baselines compared to Python, suggesting a more significant distribution gap between synthetic and human-written C++ code.

Detecting Revised Synthetic Code. In real-world scenarios, developers may revise synthetic code before using it, raising the question of whether zero-shot synthetic code detectors remain effective after revisions. We focus on the

Dataset	Code Contest C++			
Generators	CodeLlama	StarChat	GPT-3.5	GPT-4
log P(x)	67.82	61.92	73.55	69.40
LogRank	59.35	57.12	66.68	62.43
Rank	54.67	52.19	60.29	54.21
Entropy	43.83	39.52	53.74	55.50
DetectGPT	62.99	60.45	75.47	64.05
Ours $m = 8$	**89.87**	**83.42**	**90.82**	**88.85**

Table 4: Detection Results on C++. All results are reported when using CodeLlama as the Detector LLM.

most straightforward code modification: identifier renaming, which doesn't alter the code's functionality and requires no understanding of its logic. To simulate this, we extracted all identifiers and randomly replaced 10%, 20%, and 50% of them with "var_i". We then evaluated the zero-shot detectors on the revised synthetic code to observe performance changes as the fraction of replaced identifiers increased. The results, shown in Figure 5, reveal that detector performance declines as the replacement fraction increases, degrading to random guessing at 50% replacement. However, our method consistently outperforms all others across all levels of replacement. This can be attributed to GPT-3.5-Turbo and StarChat's tendency to restore "var_i" to variable names that contain code semantics, resulting in a lower similarity between the rewritten code and the revised synthetic code compared to the similarity between the rewritten code and the original synthetic code. One possible straightforward solution is initially using Code LLMs to restore all variable names and then applying our algorithm to detect the recovered code. We consider detection in this or more complex adversarial scenarios as a focus for future work.

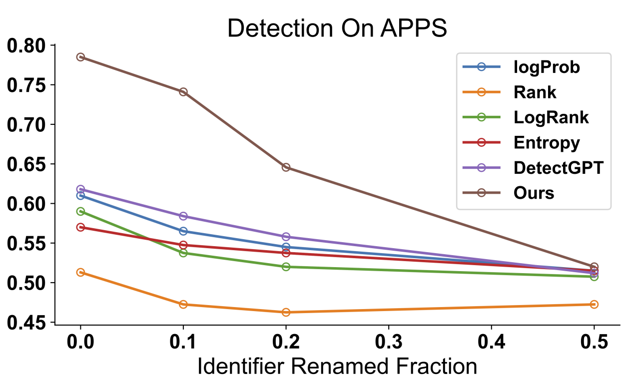

Figure 5: Detecting Revised Synthetic Code.

Impact of Code Correctness. We consider that the distribution of correct code solutions likely differs from incorrect ones since the correct synthetic code may much closer to human-written code, making it more difficult to detect. To explore the impact of code correctness, we separately present the detection AUROC for correct and incorrect codes on the MBPP benchmark using CodeLlama as Detector LLM, as shown in Figure 6. The results indicate that detecting correct code is more challenging than detecting incorrect code, both with our method and baselines. Nonetheless, our method consistently outperforms baselines when dealing with correct code.

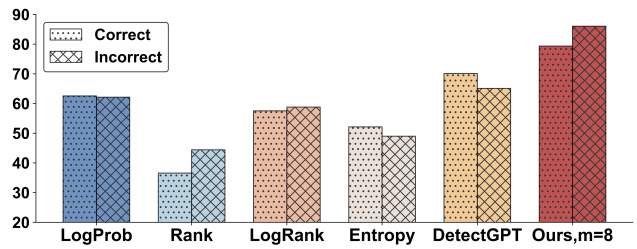

Figure 6: Impact of code correctness.

Choice of m. In our primary experiment, we set the number of rewriting m in the range of $[2, 4, 8]$ due to limited computational resources. However, increasing m can reduce randomness in code sampling and enhance the accuracy of expectation estimation. To investigate this, we conducted experiments by setting the maximum value of m to 32 on the APPS and MBPP benchmarks and plotted the AUROC against changes of m in Figure 7. The result indicates that the detection performance increases with m with slight fluctuation and saturates around 20 - 32 rewrites.

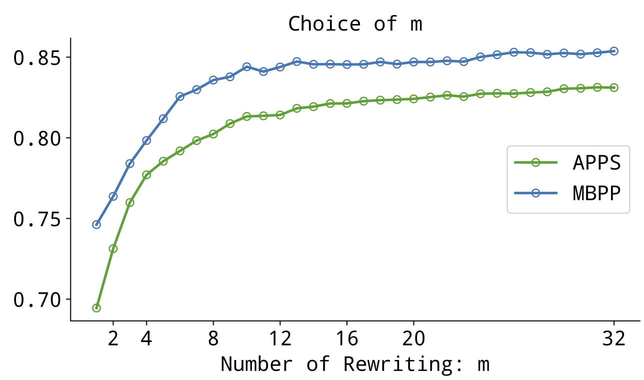

Figure 7: Choice of m. CodeLlama is used as Detector LLM.

Conclusion

In this paper, we identified the limitations of applying synthetic content detectors designed for general text to code domain and proposed a novel zero-shot synthetic code detector based on code rewriting and similarity measurement. Our approach leverages the similarity between rewritten and original code as a key indicator for detecting synthetic code. This method effectively addresses the challenges posed by the prevalence of "low-entropy" tokens in the code domain. Extensive experiments and analyses demonstrate the superior performance and robustness of our method compared to other detectors in identifying synthetic code.

Acknowledgments

This work is supported by the Key R&D Program of Ningbo under Grant No.2024Z115, partly supported by NSFC under No. 62102360 and the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform).

References

Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2024. GPT-4 technical report. arXiv:2303.08774.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

Bakhtin, A.; Gross, S.; Ott, M.; Deng, Y.; Ranzato, M.; and Szlam, A. 2019. Real or Fake? Learning to Discriminate Machine from Human Generated Text. arXiv:1906.03351.

Bubeck, S.; Chandrasekaran, V.; Eldan, R.; Gehrke, J.; Horvitz, E.; Kamar, E.; Lee, P.; Lee, Y. T.; Li, Y.; Lundberg, S.; Nori, H.; Palangi, H.; Ribeiro, M. T.; and Zhang, Y. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Denny, P.; Prather, J.; Becker, B. A.; Finnie-Ansley, J.; Hellas, A.; Leinonen, J.; Luxton-Reilly, A.; Reeves, B. N.; Santos, E. A.; and Sarsa, S. 2024. Computing education in the era of generative AI. *Communications of the ACM*, 67(2): 56–67.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Cohn, T.; He, Y.; and Liu, Y., eds., *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. Online: Association for Computational Linguistics.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, W.-t.; Zettlemoyer, L.; and Lewis, M. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Gao, T.; Yao, X.; and Chen, D. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 6894–6910. Association for Computational Linguistics.

Gehrmann, S.; Strobelt, H.; and Rush, A. 2019. GLTR: Statistical Detection and Visualization of Generated Text. In Costa-jussà, M. R.; and Alfonseca, E., eds., *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 111–116. Florence, Italy: Association for Computational Linguistics.

Github. 2021. Github Copilot. https://copilot.github.com. Accessed: 2018-12-06.

GPTZero. 2023. GPTZero. https://gptzero.me/. Accessed: 2023-12-06.

Greene, R.; Sanders; Weng, L.; and Neelakantan, A. 2022. New and improved embedding model.

Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; and Yin, J. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In Muresan, S.; Nakov, P.; and Villavicencio, A., eds., *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7212–7225. Dublin, Ireland: Association for Computational Linguistics.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; LIU, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; Tufano, M.; Deng, S. K.; Clement, C.; Drain, D.; Sundaresan, N.; Yin, J.; Jiang, D.; and Zhou, M. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

Guo, D.; Zhu, Q.; Dejian Yang, Z. X.; Kai Dong, W. Z.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; Luo, F.; Xiong, Y.; and Liang, W. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence.

He, J.; and Vechev, M. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 1865–1879.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; and Steinhardt, J. 2021. Measuring Coding Challenge Competence With APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Ippolito, D.; Duckworth, D.; Callison-Burch, C.; and Eck, D. 2020. Automatic Detection of Generated Text is Easiest when Humans are Fooled. In Jurafsky, D.; Chai, J.; Schluter, N.; and Tetreault, J., eds., *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 1808–1822. Online: Association for Computational Linguistics.

Kazemitabaar, M.; Hou, X.; Henley, A.; Ericson, B. J.; Weintrop, D.; and Grossman, T. 2023. How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, 1–12.

Li, X.; Guo, D.; Gong, Y.; Lin, Y.; Shen, Y.; Qiu, X.; Jiang, D.; Chen, W.; and Duan, N. 2022a. Soft-Labeled Contrastive Pre-Training for Function-Level Code Representation. In Goldberg, Y.; Kozareva, Z.; and Zhang, Y., eds., *Findings of the Association for Computational Linguistics: EMNLP 2022*, 118–129. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; Hubert, T.; Choy, P.; de Masson d'Autume, C.; Babuschkin, I.; Chen, X.; Huang, P.-S.; Welbl, J.; Gowal, S.; Cherepanov, A.; Molloy, J.; Mankowitz, D. J.; Sutherland Robson, E.; Kohli, P.; de Freitas, N.; Kavukcuoglu, K.; and Vinyals, O. 2022b. Competition-level code generation with AlphaCode. *Science*, 378(6624): 1092–1097.

Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; and Stoyanov, V. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692.

Mao, C.; Vondrick, C.; Wang, H.; and Yang, J. 2024. Raidar: geneRative AI Detection viA Rewriting. In *The Twelfth International Conference on Learning Representations*.

Mitchell, E.; Lee, Y.; Khazatsky, A.; Manning, C. D.; and Finn, C. 2023. DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature. In Krause, A.; Brunskill, E.; Cho, K.; Engelhardt, B.; Sabato, S.; and Scarlett, J., eds., *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, 24950–24962. PMLR.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2019. gpt-2-output-dataset.

OpenAI. 2022. Introducing ChatGPT.

Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; and Karri, R. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, 754–768. IEEE.

Pu, J.; Sarwar, Z.; Abdullah, S. M.; Rehman, A.; Kim, Y.; Bhattacharya, P.; Javed, M.; and Viswanath, B. 2023. Deepfake text detection: Limitations and opportunities. In *2023 IEEE Symposium on Security and Privacy (SP)*, 1613–1630. IEEE.

Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; Kozhevnikov, A.; Evtimov, I.; Bitton, J.; Bhatt, M.; Ferrer, C. C.; Grattafiori, A.; Xiong, W.; Défossez, A.; Copet, J.; Azhar, F.; Touvron, H.; Martin, L.; Usunier, N.; Scialom, T.; and Synnaeve, G. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950.

StackOverflow. 2023. Developer sentiment around AI/ML.

Su, J.; Zhuo, T. Y.; Wang, D.; and Nakov, P. 2023. Detectllm: Leveraging log rank information for zero-shot detection of machine-generated text. *arXiv preprint arXiv:2306.05540*.

Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; Rodriguez, A.; Joulin, A.; Grave, E.; and Lample, G. 2023a. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; Bikel, D.; Blecher, L.; Ferrer, C. C.; Chen, M.; Cucurull, G.; Esiobu, D.; Fernandes, J.; Fu, J.; Fu, W.; Fuller, B.; Gao, C.; Goswami, V.; Goyal, N.; Hartshorn, A.; Hosseini, S.; Hou, R.; Inan, H.; Kardas, M.; Kerkez, V.; Khabsa, M.; Kloumann, I.; Korenev, A.; Koura, P. S.; Lachaux, M.-A.; Lavril, T.; Lee, J.; Liskovich, D.; Lu, Y.; Mao, Y.; Martinet, X.; Mihaylov, T.; Mishra, P.; Molybog, I.; Nie, Y.; Poulton, A.; Reizenstein, J.; Rungta, R.; Saladi, K.; Schelten, A.; Silva, R.; Smith, E. M.; Subramanian, R.; Tan, X. E.; Tang, B.; Taylor, R.; Williams, A.; Kuan, J. X.; Xu, P.; Yan, Z.; Zarov, I.; Zhang, Y.; Fan, A.; Kambadur, M.; Narang, S.; Rodriguez, A.; Stojnic, R.; Edunov, S.; and Scialom, T. 2023b. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288.

Tunstall, L.; Lambert, N.; Rajani, N.; Beeching, E.; Le Scao, T.; von Werra, L.; Han, S.; Schmid, P.; and Rush, A. 2023. Creating a Coding Assistant with StarCoder. *Hugging Face Blog*. Https://huggingface.co/blog/starchat.

Uchendu, A.; Le, T.; Shu, K.; and Lee, D. 2020. Authorship attribution for neural text generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 8384–8395.

Wang, X.; Wang, Y.; Mi, F.; Zhou, P.; Wan, Y.; Liu, X.; Li, L.; Wu, H.; Liu, J.; and Jiang, X. 2021. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation. arXiv:2108.04556.

Wang, X.; Wang, Y.; Wan, Y.; Wang, J.; Zhou, P.; Li, L.; Wu, H.; and Liu, J. 2022. CODE-MVP: Learning to Represent Source Code from Multiple Views with Contrastive Pre-Training. In Carpuat, M.; de Marneffe, M.-C.; and Meza Ruiz, I. V., eds., *Findings of the Association for Computational Linguistics: NAACL 2022*, 1066–1077. Seattle, United States: Association for Computational Linguistics.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; ichter, b.; Xia, F.; Chi, E.; Le, Q. V.; and Zhou, D. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In Koyejo, S.; Mohamed, S.; Agarwal, A.; Belgrave, D.; Cho, K.; and Oh, A., eds., *Advances in Neural Information Processing Systems*, volume 35, 24824–24837. Curran Associates, Inc.

Ye, F.; Zhou, S.; Venkat, A.; Marcus, R.; Tatbul, N.; Tithi, J. J.; Hasabnis, N.; Petersen, P.; Mattson, T.; Kraska, T.; Dubey, P.; Sarkar, V.; and Gottschlich, J. 2021. MISIM: A Neural Code Semantics Similarity System Using the Context-Aware Semantics Structure. arXiv:2006.05265.

Zellers, R.; Holtzman, A.; Rashkin, H.; Bisk, Y.; Farhadi, A.; Roesner, F.; and Choi, Y. 2019. Defending against neural fake news. *Advances in neural information processing systems*, 32.

Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Wang, Z.; Shen, L.; Wang, A.; Li, Y.; et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

Zhong, W.; Tang, D.; Xu, Z.; Wang, R.; Duan, N.; Zhou, M.; Wang, J.; and Yin, J. 2020. Neural Deepfake Detection with Factual Structure of Text. In Webber, B.; Cohn, T.; He, Y.; and Liu, Y., eds., *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2461–2470. Online: Association for Computational Linguistics.