Zero-Shot Detection of LLM-Generated Code via Approximated Task Conditioning

Maor Ashkenazi[1,2⋆] (✉), Ofir Brenner[3⋆], Tal Furman Shohet[4], and Eran Treister[1,2]

[1] Department of Computer Science, Ben-Gurion University of the Negev
maorash@post.bgu.ac.il
[2] Data Science Research Center, Ben-Gurion University of the Negev
[3] Tel-Aviv University
[4] Deep Instinct

Abstract. Detecting Large Language Model (LLM)-generated code is a growing challenge with implications for security, intellectual property, and academic integrity. We investigate the role of conditional probability distributions in improving zero-shot LLM-generated code detection, when considering both the code and the corresponding task prompt that generated it. Our key insight is that when evaluating the probability distribution of code tokens using an LLM, there is little difference between LLM-generated and human-written code. However, conditioning on the task reveals notable differences. This contrasts with natural language text, where differences exist even in the unconditional distributions. Leveraging this, we propose a novel zero-shot detection approach that approximates the original task used to generate a given code snippet and then evaluates token-level entropy under the *approximated task conditioning (ATC)*. We further provide a mathematical intuition, contextualizing our method relative to previous approaches. ATC requires neither access to the *generator LLM* nor the original task prompts, making it practical for real-world applications. To the best of our knowledge, it achieves state-of-the-art results across benchmarks and generalizes across programming languages, including Python, CPP, and Java. Our findings highlight the importance of task-level conditioning for LLM-generated code detection. The supplementary materials and code are available at https://github.com/maorash/ATC, including the dataset gathering implementation, to foster further research in this area.

Keywords: LLMs · Synthetic text detection · Synthetic code detection

1 Introduction

Large Language Models (LLMs) such as Claude [1] and GPT [2] have demonstrated remarkable capabilities in text generation, excelling in tasks such as

⋆ Equal contribution.

summarization, translation, and creative writing. These models typically leverage the transformer architecture [26] and large-scale pretraining to produce coherent text. However, their broad adoption has raised concerns about misinformation and other ethical challenges [34, 8, 13], highlighting the need for robust detection methods. More recently, LLMs have shown impressive proficiency in code generation, with models like CodeLlama [20], and StarCoder [25] producing functional code snippets. These advancements transformed software development by automating repetitive coding tasks, assisting with debugging, and even generating novel solutions from high-level descriptions. Furthermore, AI coding agents and integrated development tools have transformed modern workflows by embedding generation capabilities directly into the programming process. Although these advances improved productivity, they also introduce concerns related to security, intellectual property, and academic integrity. As a result, distinguishing LLM-generated code from human-written code is crucial for mitigating potential risks. While significant progress has been made in detecting LLM-generated natural language text, identifying LLM-generated code remains a challenging problem. Prior research attributes this difficulty to the structured nature of code, which results in lower predictive token entropy compared to natural language [32]. Unlike natural language, where lexical choices and sentence structures vary widely, programming languages impose strict syntactic and semantic rules, making token probability distributions less informative for detection. In our research, we take a different approach by analyzing the role of task conditioning in improving detection. To investigate this, we conduct an initial experiment comparing LLM-generated and human-written content across natural language and code. We use two datasets: MBPP [5], which consists of programming tasks and code snippets, and WritingPrompts [12], which contains natural language stories. We use the entire test set from MBPP and sample an equivalent amount of texts from WritingPrompts, generating responses using CodeLlama for MBPP and LLaMA 3.1 [4] for WritingPrompts. To avoid unwanted effects of response lengths, responses shorter than 200 characters are discarded, while longer ones are truncated. For each response, we compute mean token entropy using the same model that generated it, under two settings: (1) unconditional sampling and (2) sampling conditioned on the original task. Extended details on the initial experiment are provided in Appendix A. The results in Figure 1 reveal a clear pattern. Without task conditioning, the entropy distributions of human-written and LLM-generated code overlap significantly, making detection difficult. In contrast, human-written and LLM-generated text from WritingPrompts exhibit greater separability even without task conditioning. When introducing task conditioning, both datasets show improved distinguishability between human-written and LLM-generated content. This finding suggests that detecting LLM-generated code without access to the task prompt is inherently challenging, but incorporating task-level context can enhance detection accuracy. Intuitively, conditioning on the task provides the LLM with specific context for the expected code, allowing the conditional distribution to primarily focus on coding style, instead of code purpose.

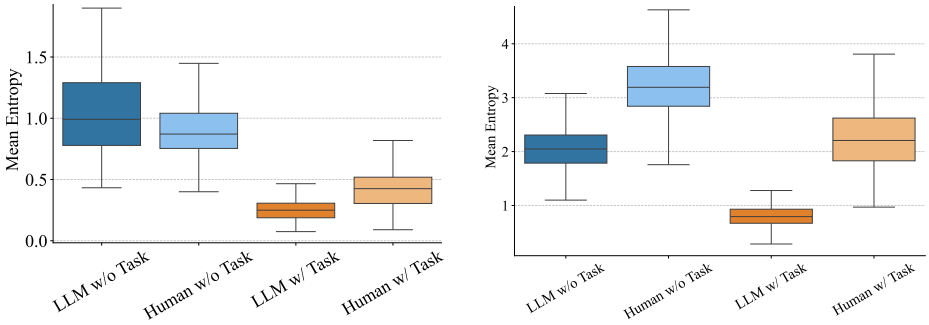

Fig. 1: Box plot of mean token entropy values for human and LLM-generated texts. MBPP (code) on the left, WritingPrompts (natural language) on the right.

Building on this, we introduce *Approximated Task Conditioning (ATC)*, a novel zero-shot detection approach that approximates a task from a given code snippet and evaluates token entropy under its conditioning. *ATC* does not require access to the original task but instead approximates it in a relatively lightweight manner, making it practical for real-world scenarios where task information is unavailable. We summarize our contributions as follows:

- We propose *ATC*, a novel zero-shot approach for detecting LLM-generated code that achieves state-of-the-art (SOTA) detection results.
- We establish a connection between *ATC* and prior detection methods.
- We conduct extensive experiments demonstrating the robustness of *ATC*.
- We release our code, including dataset gathering implementation, fostering collaboration and further research in the field.

2 Related Work

Detecting LLM-Generated Text As LLMs become more widespread, distinguishing between human and LLM-generated text becomes increasingly important to combat issues like fake news and plagiarism [34, 8]. Detection methods can generally be divided into two categories: supervised learning and zero-shot approaches. Supervised learning techniques involve training models to differentiate between human-written and LLM-generated text [35, 33, 17]. These methods often struggle with generalization, as they tend to overfit to specific datasets or the LLMs used for text generation [6, 18, 16]. In contrast, zero-shot approaches have shown greater robustness, focusing on analyzing token distribution patterns like entropy [24], likelihood [10], and ranks [22] to detect signs of LLM-generated text. Perturbation-based methods like DetectGPT [16] and NPR [22] perturb text and analyze differences between the original and perturbed texts. Similarly, [30] generates completions and compares their similarity to the original text. These methods have higher computational costs due to repeated iterations, whereas FastDetectGPT [7] improves efficiency by optimizing the perturbations.

Detecting LLM-Generated Code Text-based detectors often struggle with code due to structural differences from natural language [14, 32]. This has led to the development of specialized detection methods. [31] adapts DetectGPT with fill-in-the-middle masking which replaces entire lines of code, while [21] uses stylistic patterns such as whitespace changes in addition to preserving code correctness after perturbations. [29] proposes a method using targeted perturbations and a fine-tuned CodeBERT model. [28] conducted a large-scale evaluation of detection methods, finding that while some generalize well, they struggle often with high-level languages and short code snippets. The most recent and, to our knowledge, state-of-the-art (SOTA) method is [32], which generates code variants via multiple rewriting prompts and measures their similarity. However, this requires numerous rewrites and training a code similarity model. In contrast, our method requires less iterations, achieving superior performance with a single LLM prompt and no additional training. Meanwhile, data collection efforts, such as [9], are providing benchmarks for future detection research.

3 Method

We consider the problem of zero-shot LLM-generated code detection. Given a code snippet x, we wish to determine whether it was generated by an LLM, or written by a human. We use an open-source *detector LLM* to evaluate token probability distributions, and do not assume access or knowledge of the *generator LLM*, used for generating the code. In addition, our approach is *zero-shot*, meaning it does not require labeled training data nor involves any training steps, resulting in robust results across *generator LLMs* and programming languages.

3.1 Approximated Task Conditioning (ATC)

Here we elaborate on our LLM-generated code detection method (ATC), consisting of two main steps. First, we approximate one or more tasks for the input code snippet by prompting an LLM, which we term the *detector LLM*. Next, we calculate the score for the given code sample by computing the mean token entropy of the conditional distribution on each of the approximated tasks. When computing the score, we only consider code tokens, i.e., we ignore the task and comment tokens' entropy. We use the same *detector LLM* for token sampling for consistency. Our approach is detailed in Algorithm 1. A visualization of the pipeline and prompt used for task approximation is in Figure 2, alongside an example input, with details and connection to previous approaches below.

Choosing the Detector LLM We consider a relatively small and open-source LLM, CodeLlama13b, alongside its smaller counterpart, CodeLlama7b, as opposed to previous methods which relied on proprietary models available via APIs to achieve good results. We show that using CodeLlama7b is enough to surpass previous methods, and that using a larger model improves performance. Furthermore, we show that these relatively small *detector LLMs* achieve robust performance across various *generator LLMs* and tasks.

Algorithm 1 Approximated Task Conditioning (ATC)

Input: x: code, N: number of approximated tasks, $\mathcal{G}$: Detector LLM (vocabulary $\mathcal{V}$), ϵ: threshold
Output: Decision: `LLM-Generated` or `Human-Written`

Query $\mathcal{G}$ with x and the prompt in Figure 2 N times to generate task descriptions $t_1, \ldots, t_N$. // **Task Approximation**
for $i = 1$ to N **do** // **Code Tokens-based Score Computation**
    Concatenate the texts t_i and x.
    Perform a forward pass through $\mathcal{G}$ to get the conditional distribution $P(x \mid t_i)$.
    Get the subsequence of m **code tokens**, excluding comments: $(x_{j_1}, .., x_{j_m}) \subseteq x$.
    Compute the score for task i by calculating mean code token entropy: $Score_i = -\frac{1}{m} \sum_{k=1}^{m} \sum_{v \in \mathcal{V}} P(v \mid x_{<j_k}, t_i) \log P(v \mid x_{<j_k}, t_i)$
end for
if $\frac{1}{N} \sum_{i=1}^{N} Score_i > \epsilon$ **then**
    return `Human-Written`
else return `LLM-Generated`
end if

Task Approximation This step is performed by querying the *detector LLM* with a fixed prompt, asking it to generate a task that, when given to an LLM, would likely produce a similar code snippet to x. The full prompt is presented in Figure 2. We use $top_p = 0.95$ and a temperature of 0.7 for sampling, similar to how we generate the code solutions for the experiments. Setting top_p will limit sampling to the most probable tokens whose cumulative probability reaches 0.95, and the temperature controls the randomness of the sampling. Additional details are in Appendix B. While a single approximated task already outperforms current SOTA, our experiments show that generating multiple tasks and averaging their corresponding scores further improves performance. In most experiments, we use $N = 1, 2, 4$ approximated tasks, where N is a hyperparameter.

Code Tokens-based Score Computation Given a code snippet x, we find the m code tokens, excluding comments $(x_{j_1}, .., x_{j_m}) \subseteq x$. Next, using an approximated task t, we compute the mean token entropy conditioned on t;

$$\frac{1}{m} \sum_{k=1}^{m} H(x_{j_k} \mid x_{<j_k}, t) = -\frac{1}{m} \sum_{k=1}^{m} \sum_{v \in \mathcal{V}} P(v \mid x_{<j_k}, t) \log P(v \mid x_{<j_k}, t) \quad (1)$$

where $\mathcal{V}$ is the set of all possible tokens in the vocabulary. Intuitively, the entropy of the distribution should be lower (i.e., the model should be more confident) for LLM-generated code, as such code is more likely to align with the *detector LLM*'s learned distribution. To obtain the final score, we average the mean token entropy over all approximated tasks $t_1, .., t_N$. *ATC* can be integrated with other baseline approaches, such as computing the mean log likelihood or analyzing mean token ranks. However, we find that our method is most effective when used alongside entropy estimation, as seen in Section 4.10.

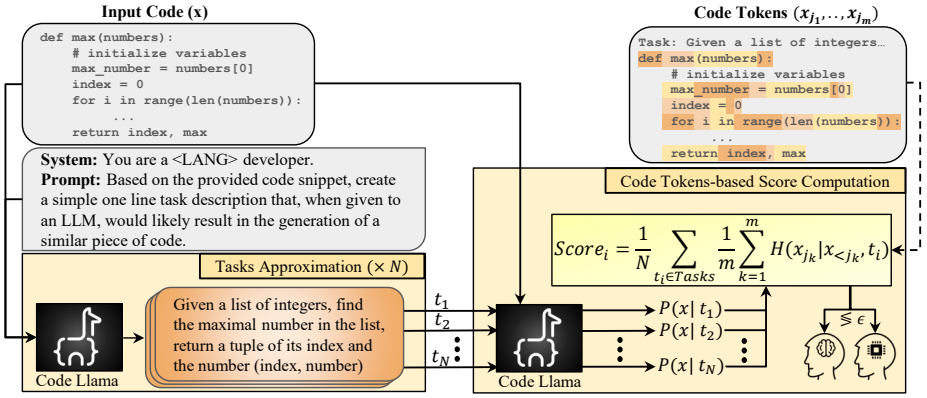

Fig. 2: **Overview of *ATC*.** Given an input code snippet x, we (1) query the *detector LLM* (CodeLlama) with a fixed prompt to generate task descriptions $t_1, .., t_N$ for which x might be a valid solution, and (2) compute the conditional entropy of the input code tokens given each approximated task. Given the probability distribution $P(x \mid t_i)$ for each task t_i, the final score is obtained by averaging the mean token entropy *only on code tokens* (see top-right). Low entropy scores indicate higher confidence that the code was LLM-generated.

Handling Comment Tokens Both human-written and LLM-generated code often include comments, such as inline comments and docstrings. We analyze the tendency to add comments in Appendix C. Due to the autoregressive nature of code generation, preceding comment tokens can significantly influence the conditional distribution of subsequent tokens. If comments accurately describe relevant parts of the code, they effectively act as a more localized and specific task within the code snippet. By treating comments this way, we aim to capture finer-grained task information, which can further aid in distinguishing between human-written and LLM-generated code. Thus, we handle comment tokens similarly to the task tokens, i.e. we exclude them from the token entropy calculation. However, *comments remain part of the input when modeling the conditional distribution.* While it is reasonable to assume that comments accurately describe relevant code, we also test robustness against adversarial modifications by evaluating the impact of removing comments before scoring (see Section 4.3).

3.2 Intuition and Connection to Previous Methods

Here we build an intuition, relating our method to zero-shot baseline methods and to [32] which we consider the current SOTA. Our observation in Figure 1 emphasizes that the predictive entropy of code tokens cannot be clearly distinguished when task conditioning is unavailable, i.e. the unconditional distributions between human-written and LLM-generated code are less separable than the conditional distributions. Baseline methods (e.g. log likelihood [10] or token entropy [24]) assess the certainty of sampling a code snippet x under the LLM's

unconditional distribution $P(x)$, since they do not have access to the task. Assuming a latent task variable $t^* \sim P_t$, the unconditional distribution of code is given by

$$P(x) = \int P(x \mid t^*) P_t(t^*)\, dt^*. \tag{2}$$

In [32], the model is prompted solely with the code snippet x and generates a rewrite x' from the conditional distribution $P(x' \mid x)$. Next, the similarity between x' and x is assessed, and high similarity leads to high confidence that x is LLM-generated. We argue that our method shares similarities with [32], particularly in how both approaches approximate the latent task t^*. Specifically, we propose that sampling from $P(x' \mid x)$ serves as an approximation to $P(x' \mid t^*)$. This is based on an assumption that conditioning on x might be similar to conditioning on t^*, as the original code x inherently carries implicit information about the underlying task t^*. Thus, one may argue an approximation of:

$$P(x' \mid x) \approx P(x' \mid t^*). \tag{3}$$

Intuitively, if a model is asked to find the maximal number in a list, its response will likely resemble the output it produces when asked to rewrite an existing snippet that does so. Sampling multiple rewrites and measuring their similarity,

$$\mathbb{E}_{x' \sim P(\cdot \mid x)} \left[S(x, x') \right], \tag{4}$$

should be correlated with the probability of x being drawn from $P(x' \mid t^*)$.

In contrast, our approach explicitly approximates the task itself $t \approx t^*$, by prompting the detector LLM. We then calculate the token entropy of x under $P(x \mid t)$. This two-step process, inspired by Chain-of-Thought principles, empirically provides a more interpretable and accurate estimation of the conditional distribution:

$$P(x \mid t) \approx P(x \mid t^*). \tag{5}$$

Finally, [32] employs multiple rewrites, which may suggest that their approximation requires multiple iterations to refine the conditional distribution estimate. In contrast, *ATC* achieves better results with a single approximation.

4 Experiments

This section details our experimental setup, covering datasets, generation models, and baseline methods. We then present the main Python results, assess the impact of comment removal as a pre-processing step, compare the approximated tasks to original ones, and analyze different task approximation prompts. We also examine robustness across factors like decoding strategies, different programming languages, and code length. Finally, we conduct relevant ablation experiments. In all experiments, we use AUROC (Area Under the Receiver Operating Characteristic curve) to evaluate performance, following previous works [32, 21, 16]. Additionally, we explore alternative metrics in Section 4.9.

4.1 Experimental Setup

Datasets To compare with the current SOTA method [32], we evaluate our approach using two widely recognized benchmarks for Python code generation: APPS [11] and MBPP [5]. To the best of our knowledge these are the most appropriate benchmarks for our task. Notably, APPS includes solutions written by a wide range of users, leading to diverse coding styles that better reflect real-world variability. APPS contains 5,000 test instances, each consisting of a problem description and corresponding solutions. After applying the data sanitization pipeline from [32], we are left with 3,765 instances. Unlike [32], which randomly sampled 1,500 instances of the test set, we use the entire APPS test set for a more comprehensive evaluation. While differences in dataset size and sampling procedures may limit direct comparisons, this approach was necessary due to the lack of code or detailed information regarding their sampling methodology. For each instance, we select the first human solution and generate corresponding LLM outputs using each *generator LLM* (detailed below). MBPP consists of Python programming problems designed for entry-level programmers, with 500 test instances. As with APPS, we use the full test set, generating LLM solutions using each generator LLM. We focus on Python due to its widespread use, readability, and versatility in code generation tasks. In both datasets we exclude the training data due to potential overlap with LLM training corpora.

Generation Models We use a variety of open-source and proprietary models for code generation. For proprietary models, we use GPT-3.5-Turbo, GPT-4o-mini [2], and Claude3-haiku [1], which were selected due to their popularity among developers [3]. For open-source models, we use Starchat-Alpha [25], CodeLlama-7B & CodeLlama-13B [20], and CodeGemma-7B [23]. These models were chosen for their widespread adoption in the open-source community. We specifically use GPT-3.5-Turbo, StarChat, and CodeLlama-13B to allow direct comparison with previous methods. For each test instance, code is generated independently by every model, resulting in a separate set of generations. We generate solutions following the schema described in [32], using Chain-of-Thought (CoT) prompting and setting $top_p = 0.95$ and the temperature to 0.7, sampling until the EOS token is reached. To ensure clean extraction, the prompt instructs the model to output the final solution between markup tags for simple parsing. Due to space constraints, results are reported using model name abbreviations.

Baselines We compare our method against several existing detection methods. First, we consider methods that estimate properties of a code sample's probability distribution using a surrogate model. This includes mean $logP(x)$ [10], LogRank and Entropy [16], and LRR [22], which combines the first two methods. Next, we look at perturbation-based methods, which estimate a code sample's properties under small modifications. We begin with DetectGPT [16] and NPR [22], adapting them to code by replacing T5_{large} [19] with $\text{CodeT5}_{\text{large}}$ [27], which, as suggested by [32], improves performance. We also consider Detect-CodeGPT [21], which builds on previous methods by applying stylistic transformations and code correctness enforcement to the perturbations. Additionally, we

Table 1: Results on MBPP.

Generator	CLlama7b	CLlama13b	Gemma	Starchat	Claude	GPT3.5	GPT4om	Avg.
$\text{OpenAI}_{\text{large}}$	52.58	49.85	26.86	39.40	31.54	49.90	40.86	41.57
Ye [32]	-	86.21	-	79.23	-	86.23	-	-
Using CodeLlama7b as Detector LLM								
DetectGPT	52.07	54.10	76.82	74.52	74.31	59.76	60.26	64.55
NPR [22]	78.20	76.09	71.27	78.34	80.95	73.46	73.37	75.95
Shi [21]	69.25	70.48	86.46	83.06	84.10	71.32	73.84	76.93
Entropy	45.68	48.28	64.47	59.58	62.38	55.16	54.75	55.76
$logP(x)$	68.37	69.79	82.78	77.85	80.72	72.95	73.29	75.11
LogRank	62.29	64.05	81.56	75.57	77.77	66.91	68.18	70.90
LRR [22]	31.20	33.67	65.80	56.57	54.29	30.72	39.41	44.52
$ATC_{N=1}$	92.82	92.62	91.20	91.18	93.82	90.47	91.23	91.91
$ATC_{N=2}$	94.06	93.60	92.67	92.10	94.85	91.82	93.02	93.16
$ATC_{N=4}$	94.36	94.25	**92.76**	92.44	95.28	92.16	93.44	93.53
Using CodeLlama13b as Detector LLM								
Entropy	46.58	50.00	63.40	58.72	61.78	56.56	55.35	56.06
$logP(x)$	68.15	71.83	82.67	77.66	81.12	74.07	74.19	75.67
LogRank	63.45	67.56	82.27	76.25	78.91	69.21	69.86	72.50
LRR	37.64	41.72	70.84	59.43	59.50	37.48	41.83	49.78
$ATC_{N=1}$	93.56	94.45	88.78	91.23	93.66	90.62	92.46	92.11
$ATC_{N=2}$	94.88	95.64	90.75	92.14	95.11	91.77	93.51	93.40
$ATC_{N=4}$	**95.94**	**96.18**	91.94	**92.74**	**95.79**	**92.62**	**94.37**	**94.22**

include results from DetectGPT4Code [31] on Java, while omitting Python comparisons as their APPS subset covered about 3.5% of the entire test set. Although we attempted to reproduce their fill-in-middle masking strategy, we observed a decrease in performance. We do not include [29] in our comparisons since the authors did not release code or sufficient details to recreate their datasets. As a supervised baseline, we include OpenAI's RoBERTa text detector [2]. Finally, we compare to [32], which we consider to be the current SOTA, as it consistently outperforms prior methods across multiple settings, including CPP. However, for MBPP, [21] achieves better results for one of the *generator LLMs*. As [32] subsampled their data ($\sim 40\%$) and code generation involves inherent randomness, we report their results as-is, acknowledging potential variability in direct comparisons due to the lack of code implementation and random seed details.

4.2 Main Results

Tables 1 and 2 present the results on MBPP and APPS, respectively. Our method outperforms all baselines across both datasets, consistently delivering significant improvements. It shows a clear advantage over perturbation-based methods, even

Table 2: Results on APPS.

Generator	CLlama7b	CLlama13b	Gemma	Starchat	Claude	GPT3.5	GPT4om	Avg.
$\text{OpenAI}_{\text{large}}$	61.11	59.24	49.01	49.76	49.84	47.33	37.43	50.53
Ye [32]	-	87.77	-	82.48	-	83.25	-	-
Using CodeLlama7b as Detector LLM								
DetectGPT	55.88	53.54	56.20	51.68	59.07	46.26	61.81	54.92
NPR [22]	62.12	60.20	59.08	55.85	68.21	53.32	60.75	59.93
Shi [21]	79.11	76.97	75.44	70.70	75.00	65.00	65.48	72.53
Entropy	47.44	46.50	56.04	46.60	63.32	53.35	49.5	51.82
$logP(x)$	67.94	66.49	72.87	60.84	73.77	63.91	54.47	65.75
LogRank	64.82	62.14	67.60	57.91	66.91	58.46	48.85	60.95
LRR [22]	46.75	40.25	39.45	42.66	31.89	34.20	29.54	37.82
$ATC_{N=1}$	92.28	93.30	91.05	88.07	92.52	86.22	87.42	90.12
$ATC_{N=2}$	93.79	94.66	92.60	89.23	94.04	88.09	89.46	91.70
$ATC_{N=4}$	94.47	95.33	93.40	89.98	94.84	89.18	90.35	92.51
Using CodeLlama13b as Detector LLM								
Entropy	41.51	41.56	53.72	43.46	59.20	54.63	44.87	48.42
$logP(x)$	62.52	65.83	72.71	59.41	72.03	65.71	51.76	64.28
LogRank	59.84	62.61	69.11	56.88	66.09	60.56	45.83	60.13
LRR	46.07	45.37	46.53	43.81	36.23	36.13	27.19	40.19
$ATC_{N=1}$	93.37	93.87	91.80	88.14	93.70	87.85	90.69	91.35
$ATC_{N=2}$	94.85	95.29	93.18	89.36	95.44	89.71	92.58	92.92
$ATC_{N=4}$	**95.62**	**96.12**	**94.10**	**90.02**	**96.33**	**90.82**	**93.70**	**93.82**

when adapted to the code domain, as well as over [32], which we consider the current SOTA. Our method remains robust, maintaining high performance across a wide range of *generator LLMs*, from smaller models like CodeLlama-7B to larger proprietary models like GPT-3.5. Notably, our approach achieves superior detection performance with just a **single approximated task** ($N = 1$), whereas [32] relied on **eight different prompts**. For a fair comparison, setting $N = 4$ further enhances performance, yielding a mean AUROC of **94.22 on MBPP and 93.82 on APPS** when using CodeLlama-13B as the detector LLM. The effectiveness of our method stems from the key observation made in Section 1: the predictive entropy of code tokens differs significantly when sampling from the conditional distribution (i.e., conditioned on the task). Our method effectively approximates the task, allowing for highly accurate detection.

4.3 Robustness to Comment Removal

To test the robustness of our method in scenarios where comments and docstrings are unavailable, we evaluate detection performance after systematically removing them from the code. Table 3 presents the results using CodeLlama-13B as the

Table 3: Results when removing comments. Detector LLM is CodeLLama13b.

Generator	CLlama7b	CLlama13b	Gemma	Starchat	Claude	GPT3.5	GPT4om	Avg.
MBPP								
Entropy	49.98	52.46	56.21	60.79	56.75	59.27	58.64	56.30
$logP(x)$	68.03	70.73	67.67	72.05	71.92	73.94	72.88	71.03
LogRank	63.33	66.18	64.78	68.89	66.89	69.30	68.30	66.81
LRR	35.75	37.02	43.77	43.04	35.64	38.09	38.97	38.90
$ATC_{N=1}$	93.92	94.68	86.29	90.59	95.09	92.76	94.51	92.55
$ATC_{N=2}$	94.82	95.50	87.69	91.65	95.88	94.01	95.18	93.53
$ATC_{N=4}$	**95.54**	**95.94**	**88.50**	**92.34**	**96.44**	**94.48**	**95.85**	**94.16**
APPS								
Entropy	46.37	45.45	56.79	51.81	61.60	56.62	53.36	53.14
$logP(x)$	59.31	60.48	70.09	60.58	72.50	67.58	56.69	63.89
LogRank	54.15	54.56	64.36	56.27	65.80	61.93	49.64	58.10
LRR	33.55	30.49	34.96	36.61	32.49	35.13	25.59	32.69
$ATC_{N=1}$	87.24	88.37	88.08	83.57	92.78	86.07	88.70	87.83
$ATC_{N=2}$	89.04	90.31	89.60	84.74	94.36	87.96	90.65	89.53
$ATC_{N=4}$	**90.68**	**91.28**	**90.72**	**85.73**	**95.28**	**89.08**	**91.89**	**90.67**

detector LLM. Our method remains highly effective, with minimal performance degradation, maintaining comparable results on MBPP and experiencing only a 3% AUROC reduction on APPS, still outperforming previous methods with $N = 1$. Increasing N further improves accuracy, reaching a mean AUROC of **94.16 on MBPP and 90.67 on APPS**. This demonstrates that our approach does not depend on comments, ensuring robustness against such transformations.

4.4 Evaluating the Approximated Task

We evaluate our approximated task by comparing detection performance against results obtained using the original task. As shown in Table 4, the mean AUROC across all *generator LLMs* indicate a slight performance drop when using the approximated task instead of the original. Figure 3 demonstrates that approximated tasks often contain slightly more detail than MBPP tasks. In contrast, for APPS, where tasks are longer and more descriptive, the approximated tasks tend to be more concise. These findings suggest that despite stylistic differences, the conditional distributions of the original and approximated tasks may still be similar. This is supported by Appendix D, which visualizes the conditional and unconditional probability distributions for the code snippets in Figure 3. Appendix E provides examples of approximated tasks from different seeds, showing how using $N > 1$ averages slight variations in the conditional distribution.

Table 4: Results using the original task with CodeLlama7b.

Method	Avg.
MBPP	
Entropy	55.76
$ATC_{N=1}$	91.91
ATC w/Task	92.02
APPS	
Entropy	51.82
$ATC_{N=1}$	90.12
ATC w/Task	92.49

Original Task

Write a python function to check whether the given two integers have opposite sign or not.

Code

```
def opposite_Signs(x,y):
    return ((x ^ y) < 0)
```

Approximated Task

Write a function that takes two arguments, x and y, and returns True if both arguments have opposite signs (positive and negative, or negative and positive), and False otherwise.

Original Task

The Rebel fleet is afraid that the Empire might want to strike back again. Princess Heidi needs to know if it is possible to assign R Rebel spaceships to guard B bases so that every base has exactly one guardian and each spaceship has exactly one …

Code

```
a,b=list(map(int, input().split..
if a==b: print("Yes")
else: print("No")
```

Approximated Task

Write a Python program that takes two integers as input and checks if they are equal. If they are equal, print "Yes", otherwise print "No".

Fig. 3: Approximated tasks examples. Left is MBPP, right is APPS. We present simple examples for readability.

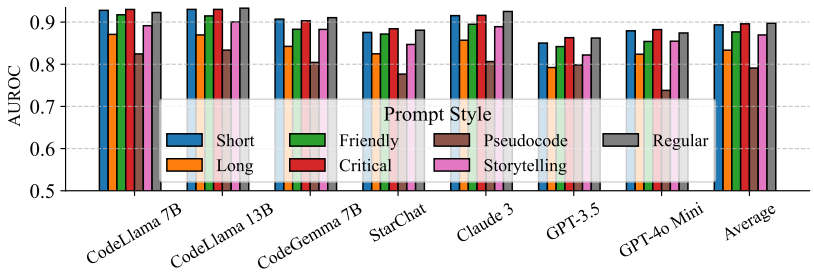

Fig. 4: ATC with different prompting styles using CodeLlama7b.

4.5 Exploring Different Task Approximation Prompts

Here we explore the sensitivity of task approximation to different prompt styles. We aim to determine whether our method remains effective across various task approximation prompts or if performance is highly dependent on specific phrasing. We test seven prompt styles, each offering a different way to approximate the task. *Regular* provides a concise task description, while *Short* enforces an even more minimal task. In contrast, *Long* generates a verbose description. *Storytelling* frames the task within a fictional scenario, *Pseudocode* translates the code into a structured pseudocode, *Friendly* offers a supportive tone, and *Critical* delivers a specific and demanding specification. In all experiments besides this one, we use the *Regular* style. Our experiment on APPS, selected for its notably descriptive tasks, reveals that prompts leading to **shorter and more accurate tasks** (*Regular*, *Short*, and *Critical*) outperform those resulting in longer tasks (*Long*, *Pseudocode*). We observed that *Storytelling* occasionally produced vague or incorrect tasks, likely due to the limitations of the relatively small detector LLM. Results are in Figure 4. Full prompts and examples are in Appendix F.

4.6 Effects of Decoding Strategies

We evaluate the robustness of *ATC* by examining the impact of decoding temperatures on detection performance, using the same configuration as in [32]. Higher temperatures introduce greater variability in the generated outputs, while lower temperatures yield more deterministic results (See Appendix B). Although entropy-based scoring methods might be sensitive to varying temperatures, the results in Figure 5 show that our performance remains robust across a range of values. Nonetheless, we do observe a slight decline in mean AUROC at higher temperatures, suggesting that high variability can impact detection accuracy.

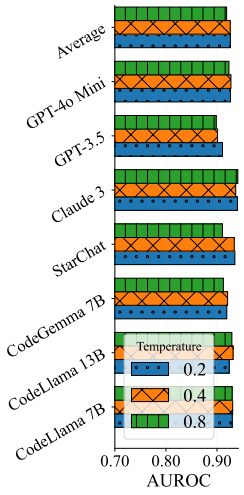

Fig. 5: Temperature effects on MBPP with CodeLlama7b.

Table 5: CodeContest results with CodeLlama7b.

Lang.	Method	CLlama13b	Starchat	GPT3.5
CPP	Shi [21]	81.07	73.59	81.96
	Ye [32]	89.87	83.42	90.82
	Entropy	29.83	39.20	43.29
	$logP(x)$	68.80	65.23	72.11
	LogRank	67.33	65.35	71.37
	LRR [22]	51.09	56.51	57.68
	$ATC_{N=1}$	97.63	90.94	92.99
	$ATC_{N=2}$	98.26	91.52	93.97
	$ATC_{N=4}$	**98.44**	**91.82**	**94.69**
Java	Shi [21]	76.65	70.72	82.10
	Yang [31]	-	-	64.03
	$ATC_{N=1}$	92.30	89.61	91.73
	$ATC_{N=2}$	92.54	90.48	93.02
	$ATC_{N=4}$	**92.93**	**90.85**	**93.00**

4.7 Generalization to Other Programming Languages

To assess the generalization of *ATC* across programming languages, which is critical for real-world applications, we experiment on CPP and Java using the CodeContest dataset [15]. We identify 152 CPP instances and 129 Java instances in the test set. Results in Table 5 focus on *generator LLMs* from previous works due to space constraints, with full results and comparisons in Appendix G. Our method consistently outperforms other approaches across all *generator LLMs*, demonstrating its ability to generalize across different programming languages.

4.8 Impact of Code Length

We examine how code length affects detection performance using APPS, where solutions are generally longer than those in MBPP. We measure length in terms

of the number of characters and group each sample—whether human-written or LLM-generated—into its corresponding length interval, independent of the original task. Consistent with previous findings, our results in Figure 7 using CodeLlama7b show that detection performance improves as code length increases, likely due to greater certainty in token predictions as the code progresses. This suggests that in practical real-world scenarios, where code is typically longer, our method is expected to perform well.

4.9 Real-World Considerations

Limitations of AUROC In practical settings, detection accuracy measured by AUROC may not fully reflect operational efficacy. Here we analyze our method's recall (true positive rate) at a fixed false positive rate (FPR). This evaluation better captures the trade-offs relevant to real-world production scenarios, ensuring reliable identification of LLM-generated code while minimizing false alarms on human-written code. As shown in Table 6, our method achieves a recall of roughly 84% at a false positive rate of 10%, demonstrating strong detection capability with minimal misclassifications. Full results are in Appendix H.

Analyzing the Number of Generated Tokens We compare the complexity characteristics of our method with the previous SOTA [32]. While both approaches rely on querying an LLM multiple times per sample, they differ significantly in the nature and length of the generated outputs. In both cases, the primary latency bottleneck lies in the generation step itself. [32] prompts the LLM with *"Please explain the functionality of the given code, then rewrite it in a single markdown code block."*. This yields a combined output containing a detailed natural language explanation followed by a full code rewrite. In contrast, our method only requires a concise task approximation. Notably, the length of this generated task remains roughly constant regardless of the input code size, compared to [32], where the output length scales linearly with the input. To quantify this, we measure the number of generated tokens on MBPP (see Figure 6). As a result, our method is not only faster but also more cost-efficient in settings that rely on third-party APIs where pricing is based on the number of generated tokens. For example, the average generation time using CodeLlama-7b per MBPP sample is **0.99** seconds in ATC, compared to **6.04** seconds in the other approach. Latency was measured on a single Nvidia RTX6000 GPU.

4.10 Ablation Experiments

Increasing the Number of Task Approximations In most experiments we use $N \leq 4$, however increasing N further enhances results. Figure 8, using CodeLlama7b, demonstrates that performance gains scale with the number of tasks, with the most significant improvement occurring at $N = 2$ and diminishing returns appearing from $N = 4$. The choice of N presents a tradeoff between accuracy and runtime, and should be adjusted based on real-world constraints.

Fig. 6: Number of generated tokens in different methods.

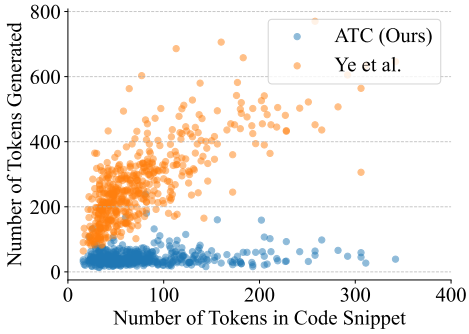

Table 6: Recall @ FPR Results.

Method	Recall @ FPR 10% Avg.	
	MBPP	APPS
CodeLlama13b as Detector LLM		
Entropy	10.09	9.96
$logP(x)$	31.23	24.58
LogRank	30.16	21.28
LRR [22]	19.18	7.01
$ATC_{N=1}$	76.12	77.72
$ATC_{N=2}$	80.37	81.58
$ATC_{N=4}$	**83.92**	**84.08**

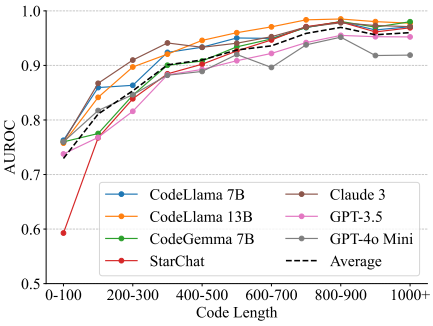

Fig. 7: Impact of code length on APPS.

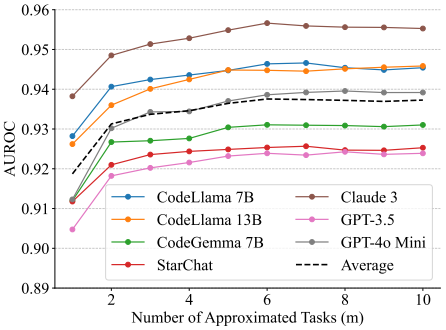

Fig. 8: Effects of increasing N on MBPP.

Comparison with Alternative Scoring Methods We replace entropy with alternative scoring methods—mean $logP(x)$, LogRank, and LRR—while keeping the task approximation framework unchanged. Table 7 presents average AUROC across all *generator LLMs*, showing that entropy is the most effective scoring method. Entropy captures global uncertainty over the full output distribution, while alternative methods rely on token-level likelihoods or ranks, making them more sensitive to local variations. Among these, LogRank performs best, indicating that ranks may provide a stronger signal than raw likelihoods. However, entropy remains the overall best scoring method across our experiments.

Score Computation with Comment Tokens We conduct an ablation study where we include comments in the token entropy calculation instead of excluding them. As shown in Table 8, this results in a consistent drop in average AUROC across all *generator LLMs*. This degradation aligns with our hypothesis that comments often serve as implicit task descriptions within the code snippet. Including them in entropy computation disrupts the separation between task conditioning and code token certainty, weakening detection results.

Table 7: Results with alternative scoring methods.

Method	AUROC Avg.	
	MBPP	APPS
CodeLlama7b as Detector LLM		
$ATC_{N=1}$ w/$logP(x)$	$80.09_{(-11.82)}$	$84.14_{(-5.98)}$
$ATC_{N=1}$ w/LogRank	$87.08_{(-4.83)}$	$76.00_{(-14.12)}$
$ATC_{N=1}$ w/LRR	$81.38_{(-10.53)}$	$80.45_{(-9.67)}$
CodeLlama13b as Detector LLM		
$ATC_{N=1}$ w/$logP(x)$	$83.94_{(-8.17)}$	$80.78_{(-10.57)}$
$ATC_{N=1}$ w/LogRank	$89.08_{(-3.03)}$	$86.06_{(-5.29)}$
$ATC_{N=1}$ w/LRR	$82.51_{(-9.60)}$	$83.07_{(-8.28)}$

Table 8: Results when including comment tokens in score calculation.

Method	AUROC Avg.	
	MBPP	APPS
CodeLlama7b as Detector LLM		
$ATC_{N=1}$	$90.10_{(-1.81)}$	$87.24_{(-2.88)}$
$ATC_{N=2}$	$91.46_{(-1.7)}$	$88.97_{(-2.73)}$
$ATC_{N=4}$	$91.91_{(-1.62)}$	$89.90_{(-2.61)}$
CodeLlama13b as Detector LLM		
$ATC_{N=1}$	$90.63_{(-1.48)}$	$88.81_{(-2.54)}$
$ATC_{N=2}$	$92.01_{(-1.39)}$	$90.57_{(-2.35)}$
$ATC_{N=4}$	$92.94_{(-1.28)}$	$91.64_{(-2.18)}$

5 Conclusion and Future Work

As LLMs become increasingly prevalent in coding tasks, their associated social and ethical risks demand reliable detection methods. We identify a key challenge: distinguishing human-written from LLM-generated code is fundamentally different from natural text when relying solely on the unconditional probability distribution. To address this, we introduce a novel, simple, and zero-shot approach that approximates the conditional probability distribution using task approximation, followed by an entropy-based scoring algorithm. Our method outperforms previous approaches across all relevant benchmarks and demonstrates robustness through extensive experiments and ablation studies. Furthermore, its simplicity enables future integration with other probability-based detection methods. Our analysis is currently limited to publicly available benchmarks; however, while our results are promising, task approximation in domain-specific repositories may pose additional challenges and warrants further study. In future work, we plan to extend our approach to detect edited LLM-generated code and explore robustness against adversarial attacks. Finally, an additional promising direction is to improve the quality of task approximations, potentially through reflective reasoning capabilities.

Acknowledgments. The authors thank the Israeli Council for Higher Education (CHE) via the Data Science Research Center and the Lynn and William Frankel Center for Computer Science at BGU.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Anthropic (2024), https://www.anthropic.com/
2. Openai (2024), https://openai.com/api
3. Stackoverflow developer survey (2024), https://survey.stackoverflow.co/2024/
4. Meta ai, llama 3.1 (2024), https://llama.meta.com/
5. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)
6. Bakhtin, A., Gross, S., Ott, M., Deng, Y., Ranzato, M., Szlam, A.: Real or fake? learning to discriminate machine from human generated text. arXiv preprint arXiv:1906.03351 (2019)
7. Bao, G., Zhao, Y., Teng, Z., Yang, L., Zhang, Y.: Fast-detectgpt: Efficient zero-shot detection of machine-generated text via conditional probability curvature. arXiv preprint arXiv:2310.05130 (2023)
8. Bommasani, R., Hudson, D.A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M.S., Bohg, J., Bosselut, A., Brunskill, E., et al.: On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258 (2021)
9. Demirok, B., Kutlu, M.: Aigcodeset: A new annotated dataset for ai generated code detection. arXiv preprint arXiv:2412.16594 (2024)
10. Gehrmann, S., Strobelt, H., Rush, A.: Gltr: Statistical detection and visualization of generated text. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations. Association for Computational Linguistics (2019)
11. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., Steinhardt, J.: Measuring coding challenge competence with APPS. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2) (2021)
12. Huang, X.Y., Vishnubhotla, K., Rudzicz, F.: The gpt-writingprompts dataset: A comparative analysis of character portrayal in short stories. arXiv preprint arXiv:2406.16767 (2024)
13. Jawahar, G., Abdul-Mageed, M., Laks Lakshmanan, V.: Automatic detection of machine generated text: A critical survey. In: Proceedings of the 28th International Conference on Computational Linguistics. pp. 2296–2309 (2020)
14. Lee, T., Hong, S., Ahn, J., Hong, I., Lee, H., Yun, S., Shin, J., Kim, G.: Who wrote this code? watermarking for code generation. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 4890–4911 (2024)
15. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al.: Competition-level code generation with alphacode. Science **378**(6624), 1092–1097 (2022)
16. Mitchell, E., Lee, Y., Khazatsky, A., Manning, C.D., Finn, C.: Detectgpt: Zero-shot machine-generated text detection using probability curvature. In: International Conference on Machine Learning. pp. 24950–24962. PMLR (2023)
17. Mitrović, S., Andreoletti, D., Ayoub, O.: Chatgpt or human? detect and explain. explaining decisions of machine learning model for detecting short chatgpt-generated text. arXiv preprint arXiv:2301.13852 (2023)
18. Pu, J., Sarwar, Z., Abdullah, S.M., Rehman, A., Kim, Y., Bhattacharya, P., Javed, M., Viswanath, B.: Deepfake text detection: Limitations and opportunities. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 1613–1630 (2023)

19. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of machine learning research **21**(140), 1–67 (2020)
20. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al.: Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)
21. Shi, Y., Zhang, H., Wan, C., Gu, X.: Between lines of code: Unraveling the distinct patterns of machine and human programmers. In: Proceedings of the 47th International Conference on Software Engineering (ICSE 2025). IEEE (2025)
22. Su, J., Zhuo, T.Y., Wang, D., Nakov, P.: Detectllm: Leveraging log rank information for zero-shot detection of machine-generated text. arXiv preprint arXiv:2306.05540 (2023)
23. Team, C., Zhao, H., Hui, J., Howland, J., Nguyen, N., Zuo, S., Hu, A., Choquette-Choo, C.A., Shen, J., Kelley, J., et al.: Codegemma: Open code models based on gemma. arXiv preprint arXiv:2406.11409 (2024)
24. Thomas, L.: Detecting fack content with relative entropy scoring. In: CEUR Workshop Proceedings, ECAI'08 Workshop on Plagiarism Analysis, Authorship Identification and Near-Duplication Detection, November. vol. 377, pp. 27–31 (2008)
25. Tunstall, L., Lambert, N., Rajani, N., Beeching, E., Le Scao, T., von Werra, L., Han, S., Schmid, P., Rush, A.: Creating a coding assistant with starcoder. Hugging Face Blog (2023), https://huggingface.co/blog/starchat-alpha
26. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
27. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (2021)
28. Xu, J., Zhang, H., Yang, Y., Cheng, Z., Lyu, J., Liu, B., Zhou, X., Yang, L., Bacchelli, A., Chiam, Y.K., et al.: Investigating efficacy of perplexity in detecting llm-generated code. arXiv preprint arXiv:2412.16525 (2024)
29. Xu, Z., Sheng, V.S.: Detecting ai-generated code assignments using perplexity of large language models. Proceedings of the aaai conference on artificial intelligence **38**(21), 23155–23162 (2024)
30. Yang, X., Cheng, W., Wu, Y., Petzold, L., Wang, W.Y., Chen, H.: DNA-GPT: Divergent n-gram analysis for training-free detection of gpt-generated text. The Twelfth International Conference on Learning Representations (ICLR) (2024)
31. Yang, X., Zhang, K., Chen, H., Petzold, L., Wang, W.Y., Cheng, W.: Zero-shot detection of machine-generated codes. arXiv preprint arXiv:2310.05103 (2023)
32. Ye, T., Du, Y., Ma, T., Wu, L., Zhang, X., Ji, S., Wang, W.: Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting. arXiv preprint arXiv:2405.16133 (2024)
33. Yu, X., Qi, Y., Chen, K., Chen, G., Yang, X., Zhu, P., Zhang, W., Yu, N.: Gpt paternity test: Gpt generated text detection with gpt genetic inheritance. CoRR (2023)
34. Zellers, R., Holtzman, A., Rashkin, H., Bisk, Y., Farhadi, A., Roesner, F., Choi, Y.: Defending against neural fake news. Advances in neural information processing systems **32** (2019)
35. Zhong, W., Tang, D., Xu, Z., Wang, R., Duan, N., Zhou, M., Wang, J., Yin, J.: Neural deepfake detection with factual structure of text. In: Conference on Empirical Methods in Natural Language Processing (EMNLP) (2020)