IS WATERMARKING LLM-GENERATED CODE ROBUST?

Tarun Suresh[1], **Shubham Ugare**[1], **Gagandeep Singh**[1,2], **Sasa Misailovic**[1]
[1]University of Illinois Urbana-Champaign, [2]VMware Research
{tsuresh3, sugare2, ggnds, misailo}@illinois.edu

ABSTRACT

We present the first study of the robustness of existing watermarking techniques on Python code generated by large language models. Although existing works showed that watermarking can be robust for natural language, we show that it is easy to remove these watermarks on code by semantic-preserving transformations.

1 INTRODUCTION

The rapid advancement of large language models (LLMs) like GPT and Codex in understanding and generating code holds transformative potential for software engineering (Chen et al., 2021; OpenAI, 2023; Ugare et al., 2024b). However, it raises concerns about misuse such as code plagiarism and malware generation. Combating misuse requires accurate detection of LLM-generated code, which is challenging as LLMs are designed to produce realistic output that mimics human-generated code.

To address this issue, researchers have developed various *watermarking techniques*, which inject hidden patterns in the generated output based on a hash or cryptographic key (Kirchenbauer et al., 2023a; Zhao et al., 2023; Kuditipudi et al., 2023). A critical challenge lies in potential human or automated modifications that can erase the patterns, undermining the watermark's detectability.

Motivation. Previous studies have shown that at least 50% of LLM-generated tokens need to be modified to remove a watermark (Kuditipudi et al., 2023). In plain text, this task is inherently challenging, requiring extensive human paraphrasing or the use of another language model (Kirchenbauer et al., 2023b). On the other hand, code is significantly easier to modify. For instance, changes to one part of a program (e.g., renaming a variable), can impact the whole program. Likewise, semantic-preserving modifications like adding dead code or employing obfuscation do not alter program behavior, enabling adversaries to easily make significant changes without compromising code quality and thereby reducing the detectability of watermarks.

This Work. We are the first to investigate the robustness of watermarking Python code generated by LLMs. We propose an algorithm that walks the Abstract Syntax Tree (AST) of the watermarked code and randomly applies semantic-preserving program modifications. We observe significantly lower true-positive rate (TPR) of detection even under simple modifications, underscoring the need for robust LLM watermarks tailored specifically for code.

Our code is available at https://github.com/uiuc-arc/llm-code-watermark.

2 ROBUSTNESS OF WATERMARKED CODE

Let x denote the sequence of tokens of length m. For an auto-regressive language model M, the objective of watermarking is to generate a watermarked completion y^w given x by embedding a hidden pattern based on a hash or cryptographic key ζ. A detector can then check if y^w is watermarked or not using ζ. A watermarking scheme consists of the following two algorithms:

- **Watermark** (M, x, ζ): Let $p_t := \mathbb{P}_{M(x)}\left[y_t = \cdot \mid y_{1:t-1}\right]$ represent the conditional probability distribution over V of the t-th token generated by M. The algorithm uses a function $\Gamma(\zeta, p_t)$ that maps ζ and p_t to a modified distribution $\hat{p}_t$ over the next token. Output y^w generated by iteratively sampling y_t^w from $\hat{p}_t = \Gamma(\zeta, p_t)$ using any of the standard decoding techniques.

- **Detect** (y, ζ): Given a completion y and key ζ, compute a p-value p with respect to the null hypothesis that y was generated independently of ζ. Return $\mathbb{1}_{p \leq p_{threshold}}$.

In practice, a user may transform a watermarked code $y^w \sim$ Watermark(M, x, ζ) into a semantically equivalent y_A such that Detect$(y_A, \zeta) = 0$. We consider that the user has only black-box

input-output access to M and has no knowledge of the watermarking algorithm, ζ, or the detection threshold. The user can apply a series of d semantic-preserving transformations $\{T_1, T_2, \ldots, T_d\}$, e.g., inserting print statements or renaming variables, to modify the code.

We replicate these realistic program modifications in Algorithm 1. The algorithm takes the watermarked code y^w and the number of transformations d to apply as input. The algorithm parses y^w to obtain the AST representation of the code. At each iteration k, a transform T_k is selected at random. The algorithm traverses the AST to determine the set of all possible insertion, deletion, or substitution locations $\mathcal{S}$ for T_k. It then transforms AST at a randomly selected $s \sim \mathcal{S}$ subtree by T_k by replacing the subsequence of terminals with a "hole" and then completing it with a random syntactically-valid sequence $\eta_k \sim \Sigma^*$.

Algorithm 1 Watermarked Program Transformation

Inputs: y^w: watermarked code, $\mathcal{T}$: set of transformations, d: number of transformations to apply

function PERTURB(y^w, d, $\mathcal{T}$)
  $AST \leftarrow$ parse(y^w)
  for $k \leftarrow 1$ to d **do**
    $T_k \sim \mathcal{T}$
    $\mathcal{S} \leftarrow$ visit(AST, T_k)
    $s \sim \mathcal{S}$; $\eta_k \sim \Sigma^*$
    $AST \leftarrow$ transform(AST, T_k, s, η_k)
  return convertToCode(AST)

3 EVALUATION

We implement several transformations: InsertDeadCode, Rename, InsertPrint, WrapTryCatch, and Mixed. Appendix A.1 presents details about our experimental setup and the transformations.

We generate Python code completions using the LlamA-7B (Touvron et al., 2023) and CodeLlamA-7B (presented in Appendix A.2) models on the HumanEval (Chen et al., 2021) dataset. We watermark the code with state-of-the-art algorithms UMD and Unigram (Zhao et al., 2023; Kirchenbauer et al., 2023a). We describe these baselines with the hyperparameter values that we used in Appendix A.3. We sequentially apply each program transformation $d = 5$ times on the watermarked code. As our transformation procedure is randomized, we run this experiment 3 times and compute the average of the results. Table 1 presents our main results for LlamA-7B. It shows that the program transformations greatly corrupt watermark detectability. Even the simplest transformations InsertPrint and Rename reduce the TPR by at least 1.3x. Complex alterations (e.g., WrapTryCatch and AddDeadCode) reduce the TPR much more significantly.

Table 1: Watermark detectability results

Algorithm	Transformation	Detection Metrics	
		TPR	FPR
UMD	Original	0.79	0
	Rename	0.57	0.01
	AddDeadCode	0.38	0.06
	InsertPrint	0.58	0.06
	WrapTryCatch	0.22	0.01
	Mixed	0.34	0.01
Unigram	Original	0.76	0.01
	Rename	0.20	0
	AddDeadCode	0.01	0
	InsertPrint	0.32	0
	WrapTryCatch	0.11	0
	Mixed	0.14	0

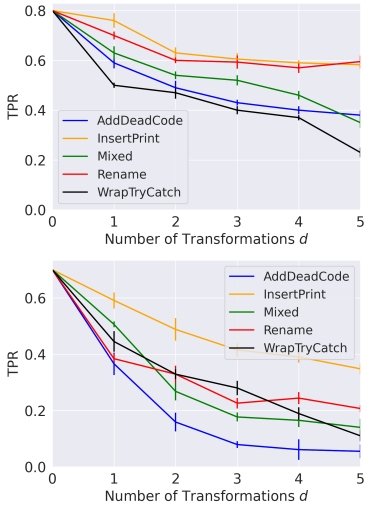

Figure 1: TPR vs the number of transformations d: UMD (above), Unigram (below).

Varying the Number of Transformations We further show the robustness of the watermark techniques by varying the number of modifications d applied to the watermarked code from 0 to 5. Figure 1 shows that the TPR declines as d increases. For instance, when employing 5 WrapTryCatch modifications, the TPR dropped to 0.22 for the UMD watermark and fell to 0.11 for the Unigram watermark. AddDeadCode and WrapTryCatch modifications exhibit a more pronounced impact on TPR, requiring fewer modifications to reduce TPR by over 2x compared to the other two modifications.

4 DISCUSSION

We are the first to study the robustness of existing watermark techniques for LLM-generated Python code. We demonstrate that realistic program modifications can easily corrupt watermark detectability. We urge future work to develop resilient detection schemes for LLM-generated code, potentially by watermarking the syntax tree of the generated code, ensuring code quality, security, and reliability in the rapidly evolving landscape of LLMs.

URM STATEMENT

The authors acknowledge that at least one key author of this work meets the URM criteria of ICLR 2024 Tiny Papers Track.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This research was supported in part by NSF Grants No. CCF-1846354, CCF-2217144, CCF-2238079, CCF-2313028, CCF-2316233, CNS-2148583, and Google Research Scholar award.

REFERENCES

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

Miranda Christ, Sam Gunn, and Or Zamir. Undetectable watermarks for language models, 2023.

Jaiden Fairoze, Sanjam Garg, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, and Mingyuan Wang. Publicly detectable watermarking for language models. Cryptology ePrint Archive, Paper 2023/1661, 2023. _urlhttps://eprint.iacr.org/2023/1661.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. A watermark for large language models. In *International Conference on Machine Learning*, 2023a.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Manli Shu, Khalid Saifullah, Kezhi Kong, Kasun Fernando, Aniruddha Saha, Micah Goldblum, and Tom Goldstein. On the reliability of watermarks for large language models, 2023b.

Rohith Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. Robust distortion-free watermarks for language models, 2023.

Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. A general construction for abstract interpretation of higher-order automatic differentiation. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022. doi: 10.1145/3563324. URL `https://doi.org/10.1145/3563324`.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. Who wrote this code? watermarking for code generation, 2023.

Linyang Li, Pengyu Wang, Ke Ren, Tianxiang Sun, and Xipeng Qiu. Origin tracing and detecting of llms, 2023.

George A. Miller. WordNet: A lexical database for English. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. Detectgpt: Zero-shot machine-generated text detection using probability curvature, 2023.

OpenAI. Gpt-4 technical report, 2023.

Edward Tian and Alexander Cui. Gptzero: Towards detection of ai-generated text using zero-shot and supervised methods, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

Shubham Ugare, Debangshu Banerjee, Sasa Misailovic, and Gagandeep Singh. Incremental verification of neural networks. *Proceedings of the ACM on Programming Languages*, 7(PLDI): 1920–1945, June 2023. ISSN 2475-1421. doi: 10.1145/3591299. URL http://dx.doi.org/10.1145/3591299.

Shubham Ugare, Tarun Suresh, Debangshu Banerjee, Gagandeep Singh, and Sasa Misailovic. Incremental randomized smoothing certification, 2024a.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: Llm generation with grammar augmentation, 2024b.

Kangxi Wu, Liang Pang, Huawei Shen, Xueqi Cheng, and Tat-Seng Chua. Llmdet: A third party large language models generated text detection tool, 2023.

Zhen Zhang, Guanhua Zhang, Bairu Hou, Wenqi Fan, Qing Li, Sijia Liu, Yang Zhang, and Shiyu Chang. Certified robustness for large language models with self-denoising, 2023.

Xuandong Zhao, Prabhanjan Ananth, Lei Li, and Yu-Xiang Wang. Provable robust watermarking for ai-generated text, 2023.

Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models, 2023.

A APPENDIX

A.1 EXPERIMENTAL SETUP

We ran experiments on a 48-core Intel Xeon Silver 4214R CPU with 2 NVidia RTX A5000 GPUs. Our program transformations are implemented using the Python LibCST library. In our experiments, we sequentially apply each program transformation d = 5 times on the watermarked code.

Let Σ denote the vocabulary of words that can be inserted or substituted into the program. In our implementation, we use the NLTK WordNet Python library for the vocabulary (Miller, 1994).

AddDeadCode A dead-code statement of the form i_1 = rand() if (i_1 != i_1): i_2 = 0 is inserted at some random location in the program where $i_1, i_2 \sim \Sigma^*$. We sample i_1, i_2 until we get a valid variable name.

Rename A single, randomly selected, formal identifier in the target program has its name replaced by a random word $i \sim \Sigma^*$. We sample i until we get a valid variable name.

InsertPrint A single print statement print(i), is inserted at some random location in the program where i is a string sequence of words such that $i \sim \Sigma^*$.

WrapTryCatch A random statement in the program is wrapped by a try-catch block.

Mixed Apply d of the aforementioned randomly selected transformations with replacement.

Table 2 shows the mean proportion of tokens changed after applying each transformation. Each AddDeadCode and WrapTryCatch transformation modifies over 20% of the tokens in the code. Consequently, in practice, it is extremely easy for the watermark to be erased away after even a few modifications by the user.

Table 2: Watermark detectability results

Transformation	Proportion of tokens changed
AddDeadCode	0.36
InsertPrint	0.12
Rename	0.05
WrapTryCatch	0.27
Mixed	0.22

Table 3: Watermark detectability results

Algorithm	Transformation	Detection Metrics	
		TPR	FPR
UMD	Original	0.82	0.01
	Rename	0.70	0.03
	AddDeadCode	0.45	0.04
	InsertPrint	0.61	0.04
	WrapTryCatch	0.28	0.01
	Mixed	0.35	0.03
Unigram	Original	0.72	0.01
	Rename	0.25	0
	AddDeadCode	0.03	0
	InsertPrint	0.26	0
	WrapTryCatch	0.09	0
	Mixed	0.12	0

A.2 EVALUATION FOR CODELLAMA

Table 3 presents the results of our evaluation on CodeLlamA-7B. Similar to LlamA-7B, we observe that the program transformations greatly reduce watermark detectability. Simple transformations like InsertPrint and Rename reduce the TPR by at least 1.4x. We observe even larger reductions for more complex modifications (e.g., WrapTryCatch and AddDeadCode).

A.3 WATERMARK BASELINES

Denote $|x|_G$ as the number of green list tokens for a generated text with length T. We experiment with the following two popular watermark schemes.

- UMD (Kirchenbauer et al., 2023a) involves selecting a randomized set of "green" tokens before a word is generated, and then biasing green tokens during sampling. Detection is performed using the *one proportion z-test*, where $z = 2\left(|x|_G - T/2\right)/\sqrt{T}$. to evaluate the null hypothesis H_0:*The text sequence is generated with no knowledge of the red list rule*.
- Unigram (Zhao et al., 2023) is proposed as a watermark robust to edit property. Detection is performed by calculating the z-statistic $z = \left(|x|_G - \gamma T\right)/\sqrt{T\gamma(1-\gamma)}$.

For both UMD and Unigram, we set γ = 0.25, where γ represents the fraction of the vocabulary included in the green list. For the Unigram watermark, we set the strength parameter $\delta = 2$, where

the larger δ is, the lower the quality of the watermarked LM output, but the easier it is to detect. We observe empirically each function completion only has around 100 tokens on average and the TPR < 0.3. To increase the number of tokens and thus the TPR, after generating the watermarked code completions, we select 3 random function completions at a time and run detection collectively on the 3 functions. We reject the null hypothesis if $z > 3$. We show that the TPR > 0.70 and the FPR ≤ 0.01 for the baseline by adopting this approach.

A.4 Related Work

LLM-generated text detection One approach to AI-generated text detection involves looking for features or statistical outliers that distinguish AI-generated text from human text. These features include entropy, perplexity, n-gram frequencies, rank, and, in the case of DetectGPT (Mitchell et al., 2023), the observation that minor perturbations of a LLM-generated text have lower log probability under the LLM on average than the original text. However, these zero-shot statistical detectors often require white-box access to model parameters, fail to detect texts generated by advanced LLMs, and rely on many text perturbations generated by another LLM, which is computationally expensive.

Another common approach is to train a binary classifier to distinguish between human and LLM-generated text. This approach assumes that LLM-generated text has distinguishing features that the trained model can identify. The fundamental problem with this is that generative models are designed with the intent of producing realistic output that is extremely hard to distinguish from that generated by humans. Specifically, recent advancements, including GPT-4 and other state-of-the-art models, are rapidly narrowing the gap between AI-generated and human-written text. As these generative models become more and more realistic, any black-box text distinguishers would incur large Type 1 and Type 2 errors. Distinguishers such as GPTZero (Tian & Cui, 2023), Sniffer (Li et al., 2023), and LMDNet (Wu et al., 2023) have no guarantee of correctness and are susceptible to issues such as out-of-distribution problems, adversarial attacks, and poisoning.

LLM Watermarking Schemes Recently, Kirchenbauer et al. (2023a) gave the first LLM watermarking scheme with formal guarantees. Their watermark divides the vocabulary into a red list and a green list based on a hash of the previous tokens and biases sampling the next token from the green list during the decoding stage. Then, a detector can count the number of green list tokens and check whether this number is statistically significant to determine whether the model output is generated without knowledge of the red-green rule.

In practice, the text generated by a language model is likely modified by a user before being fed to a detector. As a result, a line of work has focused on designing robust watermarks for text that are detectable even if the original LLM output was changed. For instance, Zhao et al. (2023) simplify the soft watermarking scheme by consistently using a fixed red-green split and demonstrate that this new watermark is twice as robust to modifications as the baseline. Kirchenbauer et al discuss more robust detection schemes for when watermarked text is embedded in a larger human-written document. Additionally, Kuditipudi et al. (2023) propose a watermarking scheme that uses a key that is as large as the LLM-generated text and then aligns that key with the text to compute an alignment cost. Recently, Christ et al. (2023) and Fairoze et al. (2023) proposed cryptographic watermarking schemes for text that achieve robustness properties.

However, these works focus mainly on watermarking LLM-generated text. They do not evaluate or provide formal guarantees on the performance of watermarks for LLM-generated code. Recently, Lee et al. (2023) proposed a new approach to watermark to LLM-generated code. They noticed that the performance of existing watermarking approaches does not transfer well to code generation tasks and attributed this to the fact that the entropy in the code generation is lower compared to that of plain text generation. They proposed a watermarking scheme called Selective Watermarking via Entropy Thresholding (SWEET) that only watermarks tokens with high enough entropy given a threshold. However, to compute the entropy of tokens at the time of detection, SWEET requires re-generating the entire code completion using the language model, which is computationally expensive.

DNN Robustness A large body of research has focused on the robustness of LLMs and other DNNs against adversarial attacks (Zou et al., 2023; Ugare et al., 2024a; Zhang et al., 2023; Laurel et al., 2022; Ugare et al., 2023). This line of work is orthogonal to our investigation as we instead focus on the robustness of whether LLM-generated output (code) can be reliably detected.