**NATIONAL RESEARCH UNIVERSITY HIGHER SCHOOL OF ECONOMICS**

Faculty of Computer Science

Касумов Джейхун Паша Оглы

Kasumov Dzheikhun Pasha Ogly

GRADUATE QUAILIFICATION WORK - MASTER THESIS

**Анализ Методов Внедрения Водяных Знаков в Языковые Модели**

**Analysis of Watermarking Methods in Large Language Models**

Qualification: Applied Mathematics and Informatics (01.04.02)

Group № мИИАД22 (Educational program “Науки о Данных/Data Science”)

|  |  |
| --- | --- |
|  | Supervisor  Sohrabi Majid  Сохраби Маджид |

Moscow, 2024

**АННОТАЦИЯ**

Эта статья посвящена анализу методов защиты больших языковых моделей (LLM) посредством исследования и применения различных постфактумных методов и методов нанесения водяных знаков, включая WLLM (A Watermark for Large Language Model), SWEET (Selective WatErmarking via Entropy Thresholding) и три новых предложенных метода нанесения водяных знаков. Анализ разделен на три части. Во-первых, изучается соответствующая литература о предыдущих исследованиях алгоритмов защиты LLM. Затем анализ направлен на последующее изучение как теоретического, так и эмпирического направлений темы исследования, причем первое предназначено для проверки и понимания фундаментальных аспектов методов обнаружения текста, генерируемых LLM, а цель второго – наблюдать за показателями производительности некоторых из рассмотренных методов нанесения водяных знаков в реальных условиях.

Проблема предотвращения несанкционированного использования и распространения текста, сгенерированного LLM, относительно недавно набрала популярность среди NLP исследователей. На данный момент, было представлено множество различных математических моделей. Эти модели можно в значительной степени разделить на постфактумные и модели генерации водяных знаков, при этом мы обнаруживаем, что последний метод более эффективен и устойчив к внешнему влиянию.

Хотя тема исследования весьма актуальна, практически ни в одной из существующих работ не рассматривается защита программного кода, генерируемого LLM. Отличительной особенностью нашей работы является тщательное изучение методов генерации и обнаружения водяных знаков в программном коде. Это могло бы позволить популяризировать соответствующую тему, дать будущим исследователям основу для анализа и дать возможность в будущем применить на практике внедренные методы по генерации и детекции водяных знаков.

**ABSTRACT**

This paper is devoted to the analysis of methods of Large Language Models (LLMs) protection through the investigation and application of various post-hoc and watermarking techniques, including WLLM (A Watermark for Large Language Model), SWEET (Selective WatErmarking via Entropy Thresholding) and three newly proposed watermarking methods. The analysis is divided into three parts. Firstly, we investigate relevant literature on previous research on LLMs protection algorithms. Then, the analysis aims at subsequently scrutinizing both theoretical and empirical directions of the research topic, where the former one is designed to inspect and understand the fundamental aspects of LLM-generate text detection methods, and the latter’s purpose is to observe the performance rates of some of the covered watermarking techniques in real-world settings.

The issue of preventing LLM-generated text from unauthorized usage and distribution has recently gained acknowledgment. A lot of different mathematical models were already introduced. These models may be largely divided into the post-hoc and watermarking streams, while we find out that the latter one is more efficient and robust to adversarial influence.

Though the research theme is quite topical, almost none of the existing papers cover protection of LLM-generated programming code – unrecognized, though extremely purposeful topic. The distinguishing feature of our work is to scrutinize methods of generating and detecting watermarks in programming code, which could allow to popularize the corresponding topic, give future researchers basis for the analysis and apply introduced LLM watermarking methods in practice.

**Table of Contents.**

[**Chapter 1: Introduction**](#_Chapter_1:_Introduction.) **5**

[**Chapter 2: Literature Review**](#_Chapter_2:_Literature) **7**

[**Chapter 3: The methodology**](#_Chapter_3:_The) **12**

[**Chapter 3.1: general theoretical background**](#_Chapter_3.1:_General) **12**

[**Chapter 3.2: text & code watermarking methods**](#_Chapter_3.2:_Text) **14**

[**Chapter 3.3: experiments and analysis**](#_Chapter_3.3:_Experiments) **25**

[**Chapter 4: Conclusion**](#_Chapter_4:_Conclusion.) **29**

[**List of References**](#_List_of_References.) **31**

[**Appendix**](#_Appendix.)**. 35**

# **Chapter 1: Introduction.**

The dawn of technological era has brought forth remarkable advancements in artificial intelligence (AI) and natural language processing (NLP). Among these advancements, Large Language Models (LLMs) have emerged as a groundbreaking development, revolutionizing the way machines understand, generate, and interact with human language. LLMs, trained on vast amounts of text data, have demonstrated unprecedented capabilities in various language-related tasks, opening up new possibilities for AI applications across industries. However, as LLMs gained widespread adoption and commercial value, the need to protect them from adversarial usage has become increasingly critical, as the unauthorized use, modification, or distribution of LLM-generated texts poses significant risks to the owners and developers of these models, as well as to the users who rely on their accuracy and reliability.

The **ultimate aim** of this paper is to scrutinize and supplement the existing LLMs protection methods, specifically setting stress on thorough theoretical & empirical analyses of existing and newly proposed methods in the conditions of low entropy settings, where the task of implementing model protection schemes becomes much more complicated. Namely, our aim is to investigate the possibilities of protecting LLM-generated *programming code*, which is a topical and purposeful, though yet unrecognized task to perform.

It is crucial to outline the appropriate stages necessary to be implemented for the purpose of attaining aim of the work:

1. Investigation of previous research conducted on this issue for the purpose of revealing past data tendencies, methodological and modelling principles, final results obtained in prior studies;
2. Theoretical analysis of the principles of language models and algorithms of LLM-generated code protection, namely *watermarking* algorithms, in the purposes of scrutinizing the fundamental principles, advantages and pitfalls of watermarking procedures in the context of LLMs output protection;
3. Practical real-world analysis of the performances of some watermarking methods, involving investigation of detectability – efficiency trade-off that is inherent to the existing models.

For the experimental (practical) analysis we apply two Python code-related datasets, namely HumanEval and MBPP, consisting of 164 and 500 test entries, each of which containing the code assignment, human-written sample solution to this assignment and several test cases. The purpose of the watermarking models is to generate watermarked Python code that will give a correct solution to the assignment (which is a prompt), at the same time yielding high detectability potential for the party that would want to check afterwards the presence of this watermark in the code. The ideal watermark should also be resilient, meaning that it should be hard to erase this watermark from the generated code.

Evidently, this research bears a substantial **theoretical relevance** to this stream of literature by thoroughly investigating yet an unrecognized, though still an extremely important topic of detection of LLM-generated programming code. This paper not only examined the existing protection methods, but also proposed three novel algorithms of implementing watermarking schemes into the machine-generated code. The results obtained in this paper may be hereafter used in the purposes of prevention and averting of adversarial exploitation of LLM-generated programming code, which may be formally considered as a **practical relevance** of the following research.

The methodology of this research covers efforts to define the “ideal” watermark, which should be detectable, resilient, inexpensive in terms of computational resources and at the same time not imposing substantial detrimental effects on the correctness and efficacy of the generated code. The best of the existing watermarks were examined both from theoretical and practical points of view, and their main advantages and especially drawbacks were outlined. In this paper, we proposed three distinct from existing watermarking schemes that aimed at resolving issues connected with previous research on the topic, and attempting to find the “perfect” watermarking method.

Before advancing to subsequent analysis, we should outline the structure of this paper. Chapter 2 refers to the investigation of previous research on the issue of protecting large language models from adversarial and malicious usage. In this section, we delve into the existing mathematical models on the post-hoc and watermarking protection streams in order to find out the most efficient one for the further examination. Chapter 3 covers theoretical and practical methodological analyses, where the former one is devoted to delving into fundamentals of language models, watermarking concept and its’ applicability in the task of protecting LLMs, while the latter analysis is performed in the purposes of estimating the performance rates of existing methods in the real-world conditions.

# **Chapter 2: Literature Review.**

Large Language Models (LLMs), such as GPT (Radford, 2018), LLaMA (Touvron et al., 2022), OPT (Zhang et al., 2022) and PaLM (Chowdhery et al., 2023), are the type of artificial intelligence (AI) models that are capable of writing texts, producing programming codes and translating documents from one language to the other, which makes them beneficial in numerous applications across various industries and domains (Liu et al., 2023). These models gained explosive popularity in recent years due to the rapidly advancing quality of text they generate, swiftly approaching them to the computer systems with human-like capabilities (Schulman et al., 2022, Zhao et al., 2023, Touvron et al., 2023).

However, as these systems become more powerful, the risk of them being used in hostile and spiteful purposes augments. These purposes include fake news dissemination (Bhat & Parthasarathy, 2020), fake product reviews (Palmer, 2023), generation of artificial training data that may spoil the performance of machine & deep learning models (Radford et al., 2022), exploitation of LLMs in plagiarism purposes (Foltynek et al., 2020), model distillations (Zhao, Wang & Li, 2023) and many other instances.

*Figure 1: Number of publications (per year) on detection of LLM-generated texts*

*(Data: Google Scholar[[1]](#footnote-1) Statistics).*

All of the abovementioned issues recently stimulated the rise of academic stream of research papers (Figure 1), all of which aim at developing the method that could allow any interested party to somehow *detect* LLM-generated text (He et al., 2022, Kuditipudi et al., 2023, Christ et al., 2023, Mitchell et al., 2023, Baldassini et al., 2024), whereas some of these papers gained huge acknowledgment: for instance, the state-of-the-art paper by Kirchenbauer et al. (2023a), which was presented in the International Conference on Machine Learning (ICML) in 2023, where it was nominated an Outstanding Paper Award, further justifying the topicality of the issue.

Still, the absolute majority of papers on the issue study detection methods on the plain text, and to the best of our knowledge there is almost no research dedicated on the detection of LLM-generated *software code*, which include code generated by different programming languages, such as Python, R, C, Java and etc. This is a huge overlook, as there is also a rocketing risk of adversarial users stealing LLM-generated code, which may lead to negative repercussions, involving code plagiarism, spreading of unreliable code solutions (which entails unpredictable code behavior, especially in sensitive applications, like security and finance), intellectual property violations (as training data often contains copyright materials) and malware generation (Chen et al., 2021, Hazell, 2023, Sandoval et al., 2023, Mirsky et al, 2023, Suresh et al., 2024). **This paper aims at analyzing the existing scarce research on the topic of detection of LLM-generated code**, additionally proposing a simple LLM-code detection method, which supplements existing knowledge on the topic of this paper.

To proceed further into the methodology and mathematical foundations of code detection methods in large language models, it is important to investigate existing research in more detail, uncovering different branches of research streams, firstly on LLM-generated plain text, and then moving to programming code detection. Overall, existing detection techniques could be divided in two branches: *post-hoc* and *watermarking* methods.

***2.1. Post-hoc detection techniques.***

Post-hoc LLM-generated text detection algorithms do not anyhow implicate in the process of text generating: these methods do not affect input data, (hyper)parameters of language model or the sampling procedure (decoding), neither they change structure or the final output of the model. Instead, post-hoc methods directly work with the original output of the model, aiming at finding some structural differences between human and machine generated texts, and trying to discover the detectable *signals* left by the model (Tan et al., 2020, Yang et al., 2023, Mitchell et al., 2023, Guo et al., 2023).

Overall, there are several mathematical-programming ideas that are fundamental in post-hoc detection schemas. For instance, there is an idea to train a new language model/fine-tune the existing one (for instance, BERT by Devlin et al. (2018)) to use it to detect LLM-generated texts (Tan et al., 2020, Ippolito et al., 2020, Muller et al., 2020). Moreover, there is also a stream of literature directed on determining statistical outliers in some parameters of the generated text, e.g., perplexity (Tian (2023) and his famous GPTZero, or Wu et al. (2023) and their LLMDet) or text probabilities (Mitchell et al. (2023) and their widely known DetectGPT).

Unfortunately, with rapid modernization and advancement of large language models, the “margin” between the human and machine generated texts swiftly disappears, thus leading to post-hoc detection methods becoming progressively less suitable in real world application purposes (Kirchenbauer et al., 2023a, Sadasivan et al., 2023, Baldassini et al., 2024). Indeed, the main advantage of these models is in that there is no need in intervening in the text generation process, which makes it possible to detect text fragments generated by all the LLMs, independently of whether these models were protected in advance or not (Kirchenbauer et al., 2023b. However, this comes at the cost of numerous pitfalls of these detection techniques, which involve necessity to know models’ parameters (white-box access), vulnerability to different kind of attacks (non-robustness), failure to detect low-entropy generated sequences, excessive computational costs and other issues that make implementation of post-hoc techniques groundless.

***2.2. Watermarking techniques.***

There is another stream of literature devoted to the detection of LLM-generated text. It is connected with the direct interference into the process of text generation via embedding of *secret signal* into the output of language model, which is imperceptible to human but allows text to be algorithmically detected as machine generated. This secret signal is called a **watermark**, and a process of imposing it into the language model is **watermarking**.

Watermarking methods on detecting machine generated texts gained tremendous popularity and recognition due to their utility and robust nature (Atallah et al., 2001, Zhang et al., 2018, He et al., 2022(a, b), Charette et al., 2022, Wang et al., 2023, Zhao et al., 2023, Baldassini et al., 2024, Suresh et al., 2024). In fact, watermarking process involves two stages: generation and detection one, where during the former stage some mathematical technique (e.g. hashing) is used by a generating party to produce the watermarked text, which makes it possible for an interested party to subsequently detect the presence of watermark in the suspicious text at the latter stage.

Overall, the idea of watermarking text goes down in history (Brassil et al., 1995). Early ideas of how to watermark texts were mostly rule-based (Atallah et al., 2001, Venugopal et al., 2011), leading to a watermark sharply decreasing the text quality. Only decades after these ideas transformed into something more statistically advanced.

Nowadays, the number of ways of imposing watermark to the generated text is surging with the recently uprising interest of researchers to investigate this field of study. One of the acknowledged ideas of efficient implementation of watermark implies biasing of next token probability distribution towards some particular set of tokens (Kirchenbauer et al., 2023a, Aaronson, 2023). For instance, the state-of-the-art paper “A Watermark for Large Language Models” (WLLM) of Kirchenbauer et al. (2023) implements the method of pseudo-random division of all tokens in vocabulary into green and red lists using some random number generator and a hashing function. After that, probabilities of all tokens in the green list are promoted by some value (hyperparameter), thus forcing language model to choose these tokens. This method gained wide acknowledgment among other researchers, and will be investigated further in this paper.

However, Kuditipudi et al. (2023) argue that the methods that involve log-probabilities biasing are at a high risk of producing low quality texts, especially in the low entropy cases. They offer a “distorted-free, agnostic and robust” watermarking technique that involves the mapping of sequence of random numbers onto a language model text via using the special watermark key. They declare high ROC-AUC values in almost all experiments, which means reaching profound detection performance while preserving decent text quality. However, they struggle with the low entropy cases, reporting worsened detection and robustness properties.

Though the concept of watermarking machine generated text is extremely prevalent now, the number of papers devoted to the analysis of watermarking LLM-generated *programming code* is astonishingly small. Imprinting watermark in the plain text is the complicated task, but watermarking programming language code is even more challenging (Zhao et al., 2023, Suresh et al., 2024). It is caused by the specific nature of these code snippets: compared to plain text, they have a significantly smaller token length, resulting in only 57 tokens on average (in human-written solutions in DS-1000, HumanEval and MBPP code datasets) (Lee et al., 2023). Furthermore, code snippets frequently suffer from a low entropy problem, meaning that it may be quite hard to rewrite the same code in the large number of ways. Finally, contrary to the plain text, programming code may be incorrectly generated (and watermarking techniques may further intensify this issue), leading to the emergence of errors. All of these issues make a qualitative watermarking code snippets an extremely complicated task, and none of the existing watermarking methods achieve high quality results in the detection of LLM-generated code.

To the best of our knowledge, only Lee et al. (2023) attempted to build the model that would impose the watermark on machine generated code snippets. In fact, they slightly modified WLLM model by Kirchenbauer et al. (2023a). In their algorithm, which was called Selective WatErmarking via Entropy Thresholding (SWEET), during the generation process they also divided the vocabulary into green and red lists, but only for tokens, for which the probability distribution reaches a certain minimum entropy (which is set as a hyperparameter), thus leading to a better performance in the context of code (and other low entropy sequences), which is expressed in rises in both the accuracy and the detectability of a generated code. However, this watermark has a lot of limitations, which will be discussed further in Chapter 3.

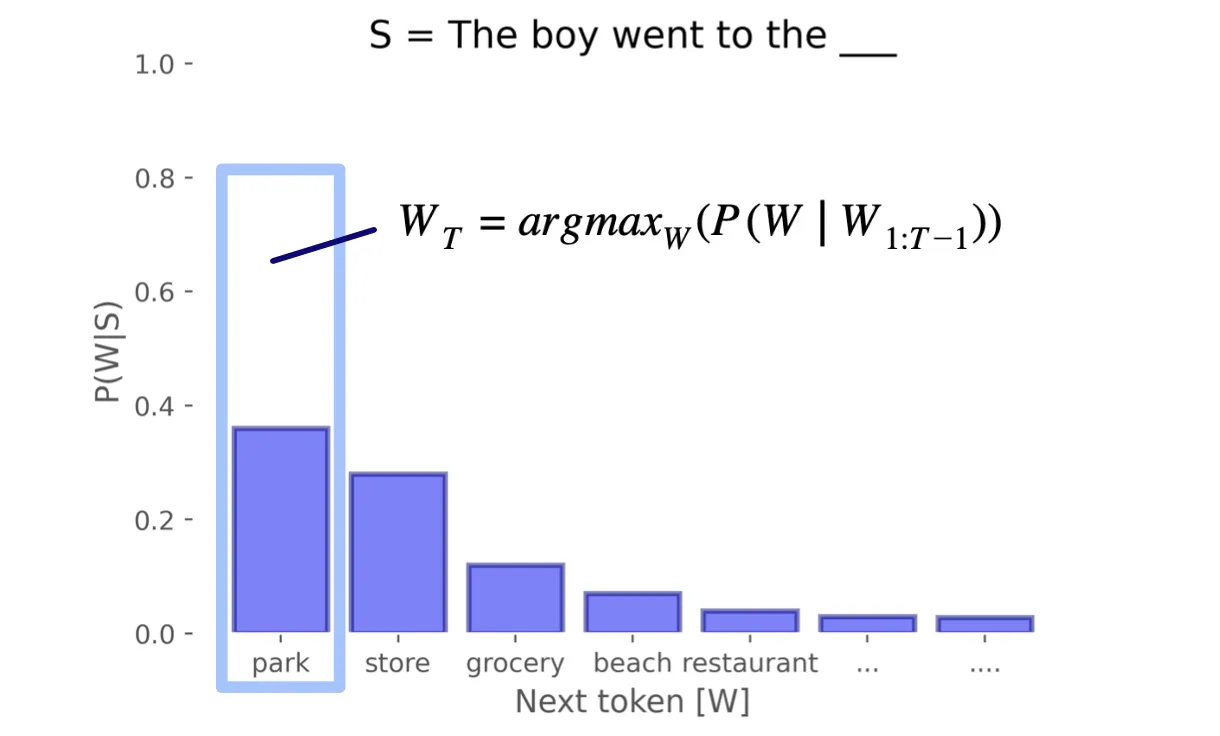
Overall, there is an extremely limited and scarce amount of research devoted to the detection of machine generated code using watermarking process. Existing watermarked generative models either fail to generate correct code or suffer from high false positive rates of watermark detection, which is a critical and inadmissible issues when analyzing watermark impositions on the LLM-generated codes. This study will explore existing code watermarking schemes in more detail, and propose one more possible algorithm of watermarking code snippets.

# **Chapter 3: The Methodology.**

In this chapter we are going to thoroughly delve into the statistical properties of a language model and a watermarking concept, analyze how it can be applied in the task of detecting machine generated programming code and discuss different advantages and pitfalls of examined watermarking algorithms. We are going to conduct certain experiments and compare the performances of different watermarking schemas in the real-world problems. This stage will include both theoretical (“vacuumed”) and practical scenarios, where the latter one will involve observing models’ behavior under various types of adversarial attacks. This practical scenario will be, in fact, the examination of watermarking methods’ resilience (robustness), which is an extremely vital property of a qualitative watermarking algorithm.

# **Chapter 3.1: General Theoretical Background.**

Let’s firstly discuss what is a language model.



*Figure 2: How LLM works: probability distribution of the next token. Source: medium.com[[2]](#footnote-2).*

***3.1.1. Large Language Models.***

Large Language Model (LLM) is an artificial intelligence model that can understand and generate human-like language, being useful in answering questions, providing explanations, generating stories, translating languages, summarizing long documents and performing other language-related tasks. Now we mathematically define LLMs. Let be a vocabulary of tokens of size . Let there be a sequence of tokens called a prompt. Let there also be a sequence of tokens {, which are already LLM-generated tokens. To generate the next token , we feed all of the existing tokens into the language model – the function, parametrized by the neural network – which in turn outputs the vector of logits of size , i.e. a logit for each token in vocabulary. Then these logits are passed through the softmax operator to turn them into the vector of discrete probabilities for each token in vocabulary (see Figure 2). Formally, language model may be demonstrated this way:

After obtaining discrete probabilities, they are used to sample the next token using one of the existing sampling (decoding) technique. For instance, greedy sampling (choosing the most probable token, see Figure 2), multinomial sampling or the technique called beam search procedure may be applied.

***3.1.2. The concept of watermarking.***

Now we examine the concept of watermarking texts from the analytical point of view.

Actually, there are different watermarking schemes, and all of them cannot be expressed in a single mathematical way. Here, we define our own watermarking method that will be introduced later in more detail. This code watermarking algorithm analyzes the pair of watermark functions (I, D), some hash value H, sequence of LLM generated tokens T and a watermark . The function I is called the implementation (of watermark) function, it takes T and and returns the watermarked sequence of tokens:

The function D is called the detector function, it takes the sequence of tokens S, some predefined hash value H and a watermark and returns the following:

However, we note that this is only one of the possible ways of imprinting the watermark into the text. Further we will also discuss algorithms that do not follow this

The ideal watermark properties were previously discussed in number of papers (Atallah et al., 2001, Kirchenbauer et al., 2023a, Kuditipudi et al., 2023). In our view, the perfect watermark should possess the following properties:

1. **Minimalized text distortion**: the implementation of watermark should not tangibly degrade the quality of generated text. The negative influence on text quality should be minimalized or even fully averted. This is especially important for code watermarking.
2. **Independence from LLM**: it should be possible to run the detection algorithm without knowing the details of a prompt or a language model that were used in generation of the text. In other words, there should be no requirements on the white-box access to LLM. Otherwise, it would be impossible to run the detection algorithm on texts generated by black-box models.
3. **No retraining** **requirement**: the process of watermark imprinting should be fairly simple and computationally affordable. There should be no necessity in retraining the language model to generate the watermark.
4. **Resilience**: the purpose of a watermark imposition is to defend generated texts from stealing by adversarial users, who certainly would try to scrub the watermark using different fraudulent schemes. Ideal watermark should be robust to such kind of adverse attacks.

Next, we will move to the analysis of some existing watermarking methods and see, whether they may be applied in the context of generating and detecting programming code.

# **Chapter 3.2: Text & Code Watermarking Methods.**

The purpose of this paper is to focus on three particular watermarking algorithms and discuss their strong and weak sides. Here, we investigate WLLM (A Watermark for Large Language Models) by Kirchenbauer et al. (2023a), SWEET (Selective WatErmarking via Entropy Thresholding) by Lee et al. (2023) and three newly proposed watermarking schemes.

***3.2.1. A Watermark for Large Language Models (WLLM).***

WLLM by Kirchenbauer et al. (2023a) propose a dynamic watermarking algorithm, which imprints watermark not after, but during the process of text generation, which is different from the watermarking technique we introduced above. Its’ main idea is in the biasing of sampling (decoding) method of generating next token via the promotion of some “green” set of tokens (see Figure 3).

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Автоматически созданное описание

*Figure 3: How WLLM’s green list promotion works (in logits).*

So, let there be a set of tokens , which consists of the prompt and already generated tokens. We feed these tokens into the language model, which outputs the vector of logits for each word in vocabulary, which are then transformed into the vector of probabilities using the softmax function. After that, using some hash function, we compute the hash of the sequence of k previous tokens (k is a hyperparameter, which is advised to be not very large) and use this hash to seed some random number generator. Now, the watermark is being imposed: using the seed, we divide the vocabulary into green list G and red list R, which may be of different sizes ( hyperparameter controls for the size of G), thus obtaining G of size and R of size . The aim of WLLM is to promote the logits of green list tokens (by hardness hyperparameter ) to bias the LLM’s output text towards these green tokens, which could be later revealed by the detection party. This promotion is depicted in Figure 3, and analytically it could be expressed via modified softmax operator function at some time moment n:

The modified probability distribution is then used to sample the next token using some decoding scheme.

Finally, the detection procedure is implemented as follows: knowing the hash function, the detector party is able to compute how many green and red list tokens there are in the text. If the text is machine generated, then the proportion of green list tokens should be substantially larger than the green list size , while a human-written text would be expected to contain approximately proportion of green tokens. Hence, a simple one-sided z – test is conducted:

In the experimental part, we will tune the z – score in order to achieve the best combination of true positive to false positive rates.

Before stating the main problem of this model in watermarking code snippets, we introduce the concept of *spike entropy with modulus z:*

Generally, classical information entropy quantifies the amount of uncertainty and randomness in a data source. In our case, spike entropy shows, how uniform at the current time moment n the probability distribution for the next token is, reaching minimal value when probability of a single token is 1 and probabilities of other tokens is 0, while achieving maximal value when the probability distribution is uniform.

Изображение выглядит как текст, Шрифт, снимок экрана, белый

Автоматически созданное описание

*Figure 4: WLLM-watermarked Python code fragments on two different prompts. Here we observe two cases of weak watermark & code correctness (above) and strong watermark & code incorrectness (below). Thus, the trade-off between accuracy and detectability may be seen.*

Изображение выглядит как текст, Шрифт

Автоматически созданное описание

Now we arrive at the main problem of WLLM. In contrast to the plain text (for which WLLM shows very decent results), code snippets usually possess significantly lower average spike entropy across all tokens (the case as in Figure 3) – the number of ways we can rewrite the same code is usually quite small. Thus, implementing a weak watermark with a lower hardness parameter leads to the problem that the largest probability token is not affected (as with token Obama in Figure 3), thus leading to the case when the accuracy of the generated code is not really affected, but at the same time the watermark is too weak to detect it – the number of green words is just too small (Figure 4, upper part). It could be possible to dramatically increase for the green tokens to have approximately the same probability as the largest probability token, but this measure leads to the dramatic decrease in the code correctness rates (the code fragment simply produces incorrect results, or even encounters errors) (Figure 4, lower part). Kirchenbauer et al. (2023a) state that WLLM in the context of low-entropy sequences could provide decent results in the case when the number of tokens is large enough. But here comes one more problem: as it was stated above, generated code snippets are usually of a much smaller length compared to plain texts (averaging 57 tokens per code snippet, which is extremely scarce).

So, a huge issue when using WLLM in generating watermarked code snippets is that this model imposes the problematic trade-off between code accuracy and code detectability. Further we analyze SWEET procedure by Lee et al. (2023), which aims at resolving this issue.

***3.2.2. Selective WatErmarking via Entropy Thresholding (SWEET).***

SWEET by Lee et al. (2023) effectively applies a very similar to WLLM procedure, except that it tries to “tune” it to the code-generation problem. The key idea of SWEET is that watermark is imprinted not on all the tokens, but only on their subset.

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

*Figure 5: In both the above and below SWEET-watermarked code fragments the number of green tokens is significantly larger than the number of red ones, which indicates high detection ability of SWEET watermarking method. Moreover, both code snippets output correct results without errors. Tokens in black are the low-entropy ones, we do not impose or detect watermark among these tokens.*

Изображение выглядит как текст, Шрифт, белый, снимок экрана

Автоматически созданное описание

Viz, for each next token we generate the probability distribution of all tokens in vocabulary. But now, we proceed further into dividing the vocabulary into G and R only if the spike entropy of the next token is larger than a certain threshold (an additional hyperparameter). In other words, we impose the watermark only on high-entropy tokens, i.e. tokens, for which WLLM procedure is the most efficient. It allows us to avoid low-entropy information, which undermines the detection abilities of WLLM model. Hypothesis testing here also applies one-sided z – test, but now we take into account only high-entropy tokens:

Here, is the total number of tokens, which entropy exceeds the threshold (green and red tokens, blacks are excluded), while is the number of green tokens among .

Let’s look at Figure 5, where SWEET-generated code snippets may be observed. Indeed, we may see that there are a lot of green tokens in these code functions, which are also correctly generated so that to produce correct output. Concurrently, black tokens of low entropy do not downgrade the detection abilities of watermark here. Thus, detectability and correctness are both maintained.

SWEET method is quite effective at first sight. However, it has a lot of critical disadvantages. Firstly, in contrast to WLLM, where there is no need to run the language model when conducting watermark detection, using SWEET requires detection party to run LLM (to get entropy values for each token in the text), which in turn demands the knowledge of the prompt that was used to generate the code snippet. In simpler words, SWEET algorithm requires a complete white-box access to LLM, which is, evidently, not realizable in the real-world conditions. Furthermore, because of the application of LLM and entropies calculation, SWEET imposes a dramatic computational burden, meaning that it is too computationally expensive (will be discussed in more details in Experiments section).

Now, let’s take a closer look, again at Figure 5. By scanning the code snippets, we may observe that the majority of green tokens are actually variables. This is logical: high entropy tokens in the programming code snippets are frequently indeed just variables, while all other tokens on the actual structure of the code are usually the low-entropy ones.

Изображение выглядит как текст, Шрифт, снимок экрана, рукописный текст

Автоматически созданное описание

*Figure 6: Variable paraphrasing attack is implemented. Code snippets still give correct results, but now the watermark is simply undetectable.*

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Let’s suppose that an adversarial user knows about the presence of watermark in the text, and tries to scrub it. He implements a very simple variable paraphrasing attack (actually, variable renaming is the very first thing that any person tries to implement). Let’s look at Figure 6. We may observe that variable renaming caused a sharp rise in the number of red tokens, leading to a drastically worsened detecting performance. The problem aggravates due to the presence of large quantity of black tokens, which reduces the total number of detectable tokens (which was already small) and makes a task of successful attacking the watermark much easier.

We infer that SWEET watermark was scrubbed under the trivial adversarial user attack, signifying the fact that SWEET turned out to be vulnerable and non-robust even under a “light” human pressure. More detailed analysis on adversarial attacks and robustness of the models will be covered further.

***3.2.3. Python Enhancement Proposal (PEP) Watermarking and other ideas.***

Surprisingly, the abovementioned two algorithms are the only ones that show fairly decent results in the generation & detection of watermarks in programming code and other low entropy scenarios. Actually, to the current moment, SWEET by Lee et al. (2023) is the only algorithm designed specifically for code watermarking. In this paper we offer some ideas of how to perform watermarking procedure in code scenarios, which could show good performance individually or in the combination with other watermarking methods.

As stated by Kirchenbauer et al. (2023a), their method WLLM struggles to correctly embed the watermark in the low entropy & short texts cases. Indeed, Python code snippets may contain a very insignificant number of tokens that could be even further lessened by the introduction of entropy thresholding and black tokens as in SWEET by Lee et al. (2023).

So, there appears a cunning question: is it possible to make a Python code longer and more verbose in order to make it more “watermarkable” and augment the performance statistics of existing methods? We arrive at **Method 1**. Formally, there indeed exist several ways to lengthen Python code fragments without affecting their correctness and efficacy. We name some of these techniques here:

1. Lengthen variable names and add more tokens into them:

*Instead of: x = 5*

*Use: assign\_the\_value\_five\_to\_the\_variable\_x = 5*

1. Import unnecessary modules and/or import modules multiple times:

*Instead of: import math*

*Use: from math import exp, trunc, ceil, sqrt, …*

1. Comment everything in more detail:

*Instead of: y = x \*\* 2*

*Use: y = x \*\* 2 #### Defining the parabola function*

1. Use more token-heavy syntax:

*Instead of: a = 3 + 5*

*Use: a = int(3 + 5)*

*Instead of: print(“Hello, world!”)*

*Use: print(“Hello” + “,“ + “ “ + “world” + “!”)*

1. Use function wrappers & unnecessary type hints:

*Instead of:*

*def add\_wrapper(a, b, c):*

*return a + b + c*

*Use:*

*def add\_wrapper(a, b, c) -> int:*

*return add\_inner\_wrapper(a, b) + c*

*def add\_inner\_wrapper(a, b) -> int:*

*return a + b*

1. Use list comprehensions with unnecessary complexity:

*Instead of: numbers = [1, 9]*

*Use: numbers = [x\*\*2 for x in range(1, 4) if x % 2 != 0]*

1. Unnecessary use of try-except blocks:

*Instead of:* *x = 5 / 2*

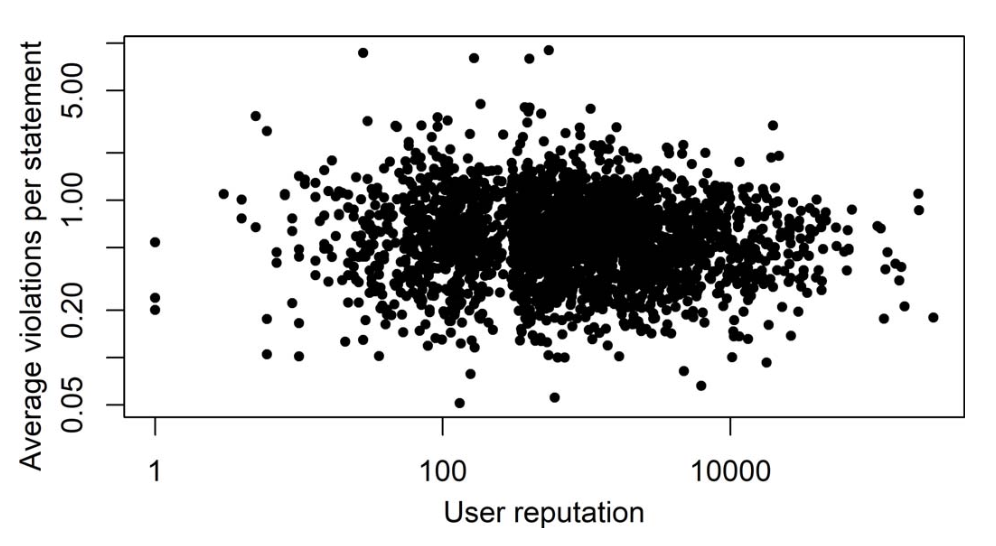
*Use:*

*try: x = 5 / 2*

*except ZeroDivisionError: pass*

We listed only seven, but in fact there are much more ways of augmenting the number of tokens in the generated code, which may allow to achieve more stable watermarking performance, not only in WLLM and SWEET, but in a lot of other algorithms, which efficiency directly depends on the number of tokens in the generated text. Though not affecting code correctness, the proposed method of increasing the number of tokens may negatively affect code readability, meaning that it should be implemented with caution, keeping the optimal trade-off between code detectability and analyzability.

The next two watermarking methods analyzed will be based on the watermark definition covered in Section 3.1.2. Now let’s arrive at **Method 2**. We introduce the concept of PEP8 (Python Enhancement Proposal) – a document that provides guidelines and best practices on how to write Python code. The main goal of PEP8 is to improve the readability and consistency of Python code.



*Figure 7: Average number of PEP8 violations in human-written Python codes per user reputation. Source: Paper by Bafatakis et al. (2019).*

We know that a lot of previous academic works outlined the fact that human-written Python codes contain a lot of PEP8-style errors. For instance, Bafatakis et al. (2019) stated that 93.87% of examined human-written codes (407,097 snippets) contained style violations, with a mean of 0.7 violations per Python statement. Moreover, in Figure 7 it may be observed that the number of violations per code statement do not correlate with the user reputation. Besides relying on the above paper, we also conducted the similar analysis on MBPP dataset (entries with human-written sample code solutions) and found out that a human makes on average 8 violations of PEP8 per a code snippet.

That is why we propose the so-called **Python Enhancement Proposal (PEP) watermark**. This watermark requires the creation of the vocabulary of all possible tokens that could be used in the generation of Python text, including tokens representing names of different libraries. Then, each token should be assigned a {secret} number: the rarer is the token, the larger number it has, representing its’ enlarged impact. In fact, it is possible to just compute IDF (Inverse Document Frequency). We note that tokens of high importance (equivalently, the tokens that are harder to remove by an attack) should be assigned a significantly larger number. By this logic, all variables are assigned with a minimal number.

The watermarking process is the following:

1. LLM generates unwatermarked code.
2. All PEP8 violations are corrected.
3. Each token is given a number corresponding to the one discussed above.
4. All numbers are summed.
5. The resulting number is used to seed the random number generator, which chooses one or several {insensible} PEP8 violations (depending on the choice of RNG) and puts them into the code.

The detection party may go through all these steps and find out, which and how many PEP8 violations should contain the following code if being machine-generated. We *emphasize* that RNG should be insensible to small and moderate changes in the seed in order to make a watermarking scheme more robust to adversarial attacks. The degree of insensibility is a hyperparameter that should be tuned.

This watermarking scheme is not resilient in the case of a user cleaning the generated Python code from PEP8 violations using some online service (for instance, autopep8[[3]](#footnote-3)). We note that false positive rate produced by this method should be quite low, because the probability of human-generated code having exactly the same PEP8 violations that were predicted by a structure of the code should not exceed some low threshold. We propose this basic watermarking idea in the hope of the future researchers to investigate it further.

We may propose one more idea for a Python code watermarking procedure, concluding with **Method 3**. In fact, it is similar to the first proposed method: we aim at reformulating code fragments in the way that they retain their logic. But now there is no need to enlarge the number of tokens in the code. Instead, we would like to embed the watermark via varying the structure of the code with the preservation of its’ logic and purposes. For instance, code fragment *x = k + n* may be formulated as *x = n + k*, thus embedding watermarkwithout the loss in code efficiency. Let’s enumerate some of the proposals for similar code alterations:

1. Reorder variables in mathematical operations:

*Instead of: x = k + n*

*Use: x = n + k*

*Instead of: x > y*

*Use: y < x*

1. Add or delete whitespaces:

*Instead of: a=b+c+d+e*

*Use: a = b + c + d + e*

1. Add or delete comment signs:

*Instead of: #### This is a function*

*Use: # This is a function*

1. Add or delete unnecessary variables:

*Instead of:*

*def f(y, z):*

*x = y + z*

*return x*

*Use:*

*def f(y, z):*

*return y + z*

1. Converting from and to list comprehensions:

*Instead of:*

*lis = []*

*for i in range(5):*

*lis.append(2 \* i)*

*Use:*

*lis = [2 \* i for i in range(5)]*

1. Swapping if and else statements:

*Instead of:*

*if x > 3: y == 5*

*else: y == 6*

*Use:*

*if x <= 3: y == 6*

*else: y == 5*

1. Create or get rid of nested if statements:

*Instead of:*

*if x == 3:*

*if y == 4:*

*print(x + y)*

*Use:*

*if x == 3 and y == 4:*

*print(x + y)*

Once again, we listed seven examples, though there may be a vast number of different applicable transformations of Python code that would not affect its’ efficiency and logic. How the watermark may be embedded in this method? Well, the simplest case is to assume one-sided possible transformations (e.g., for loops always transform to list comprehensions, but not vice versa) performed on the already generated code in the process of watermark imprinting. Uncertain cases, e.g., when changing order of variables in math operations, may be resolved via using some secret hash function that would compute hash of two variables and always put a variable with a larger/lower hash value at the first place. The main drawback of this “one-sided” approach is that it may potentially lead to a higher false positive rate: if, for instance, watermark is embedded via transforming for loop into list comprehension, it may occur that list comprehension may also be used by a human in a human-written code, which may lead to erroneous detection of a watermark in the human-written code. However, this issue may be averted if a lot of different transformations are applied to the code, and it is stated that a suspicious code is concluded to be a machine-generated one if and only if at least some percentage (a hyperparameter) of a possible number of transformations were detected in the code.

This method is perspective, though it is not perfect because it may be vulnerable to a certain type of paraphrasing attacks. We urge future researchers to examine the performance of this method, also including its’ “two-sided”, more intricate, version. Now we move to the experimental section, where we will observe the behavior of the watermarking methods in the real-world conditions.

# **Chapter 3.3: Experiments and Analysis.**

Next, we are going to analyze the performance of the watermarking algorithms investigated in Chapter 3.2 in theoretical (“vacuumed”) and real-world (“attacked”) conditions, where in the latter scenario, we will introduce some most popular watermark attacking and code paraphrasing techniques. But firstly let’s cover the technical details and information on datasets that were applied in the research.

***3.3.1. Information on datasets & Technical details.***

This research aims at scrutinizing the performance of watermarking methods applied on programming code, meaning that we will need code datasets. In this paper we decided to choose HumanEval[[4]](#footnote-4) (Chen et al., 2021) by Open AI and MBPP[[5]](#footnote-5) (Mostly Basic Python Problems Dataset) (Austin et al., 2021) by Google Research code datasets, which consist of the programming task (prompt), human-written (canonical) solution and test cases to check the correctness of the generated code. The example of HumanEval entry may be found in Appendix.

The LLM used to generate watermarked sequences is StarCoder[[6]](#footnote-6), the state-of-the-art language model that specializes on programming code generation.

The analyzed programming language is Python 3. All the necessary computations were carried out using Google Colab and Yandex Cloud servers.

Generating watermarked text requires a strong computational power. We used GPU NVIDIA Ampere A100 with 200 GB of CPU and 80 GB of GPU memory.

All the necessary Python codes and the corresponding files may be found in this paper GitHub repository: https://github.com/killiganni/LLM-Watermarking. We thank Lee et al. (2023) for providing code, necessary to perform the SWEET procedure.

***3.3.2. Experiments on Correctness & Detectability.***

We conducted several experiments on WLLM and SWEET watermarking methods to observe their performance in real-world settings. Let’s look at Table 1. Here we may observe correctness and detectability of WLLM and SWEET for two datasets (HumanEval and MBPP) under different hyperparameters, which are indicated in parentheses next to the name of the method: for WLLM, it is gamma (green list size) and delta (hardness parameter), while SWEET also includes the third hyperparameter – epsilon, which shows the entropy threshold.

The choice of the hyperparameters is not random – we included only those ones, which were recommended by authors of the corresponding papers. Moreover, the choice of hyperparameters is based on the aspiration to cover all kinds of watermarks, namely weak, moderate and strong. The logic is the following: the larger is gamma and the smaller is delta, the weaker is the watermark, though the better are the code correctness results (due to a smaller influence of the embedded watermark). For SWEET, we chose epsilon variants that maximized the detectability performance of the model, while not significantly harming the quality of the code generated.

Изображение выглядит как текст, чек, снимок экрана, Шрифт

Автоматически созданное описание

*Table 1: Experiment results of watermarking methods. We analyzed correctness and detectability of LLM-generated code under different watermarking settings.*

So, in Table 1 the results on the code correctness (Pass@1) and code detectability (ROC-AUC, TPRs) may be observed. Here, Pass@1 (Chen et al., 2021) estimates the percentage of code correctly performing, while TPR@0%, TPR@1% and TPR@5% show true positive detection rates at different levels of false positive rates. Once again, the methods performance was assessed on two code datasets – HumanEval (with 164 entries) and MBPP (with 500 test entries). We also included the Pass@1 correctness rate for the code generated without any watermarking schemes, which was done in the purposes of estimating the influence that watermarking methods have on the correctness & efficiency of the code generated.

Overall, the experiment results are controversial.

On one hand, it may be seen that SWEET *indeed* shows substantially better detectability results compared to WLLM (at least in the examined low entropy code settings): almost in each case and for both datasets, SWEET ROC-AUC levels are higher than those of WLLM. In almost all moderate-strong watermarking cases, SWEET ROC-AUC rates are higher than 0.9, which are quite decent detectability results. Moreover, the most interesting TPR@1% rates are substantially higher in SWEET settings, which indicates that in moderate and strong scenarios this watermarking scheme is able to detect 18.4, 50.6, 54.3 and 69.4 % of watermarked sequences with just a 1 % false positive rate. Meanwhile, these rates for WLLM are just 9.2, 15.9, 24.4 and 40.6 %, which is considered to be quite an ineffective result, meaning that WLLM poorly deals with detection of watermarks in the machine-generated code problem. We also note that differences in Pass@1 correctness rates for both methods are overall insignificant, while SWEET showing higher correctness rates for a stronger watermarking scheme, and WLLM being more effective in the moderate scenarios.

Concurrently, there are substantial pitfalls in the performance of both methods. While we generally confirm the detectability (ROC-AUC, TPRs) results obtained in SWEET paper, we see that the code correctness rates after applying WLLM and SWEET are in fact substantially smaller than those indicated in SWEET paper. We observe that implementing moderate and strong watermarks indeed affects Pass@1 rates: the decline in correctness levels accounts for 10.5 – 42 % for WLLM and 14.1 – 36.8 % for SWEET (under different hyperparameter conditions). We note that in fact correctness rates could be preserved if embedding weak watermarks – the fall in Pass@1 accounts for just 5.2 and 8.7 % for HumanEval and MBPP for both methods. But at the same time, it is evident from the Table 1 that weak watermarks show *very poor* detectability results of generated codes, which account for just 0.6 – 0.65 ROC-AUC levels, and a catastrophic 1.2 – 2.8 % TPR@1% rates, which is an extremely low result. Hence, we infer that the obtained results demonstrate evident trade-off between detectability and correctness rates, being a serious problem especially in MBPP settings, where this trade-off results in choosing from detectable & ineffective and undetectable & effective watermarking schemes.

Overall, we see that even under a “safe” and “unattacked” setting, the performance of both WLLM and SWEET methods is controversial. We confirm that SWEET, in general, shows better results (especially in detectability rates) in the code and other low entropy settings. Nevertheless, as was discussed earlier, SWEET is in fact extremely non-resilient to different kinds of human attacks, especially the paraphrasing ones, which makes this method impractical in real-world conditions. In section 3.2.3 we proposed three other watermarking methods, which may be used to imprint watermarks into the code snippets, where Method 1 may be used to reinforce the performance of both WLLM and SWEET, and Methods 2 and 3 being very useful in the conditions, where we need to fully preserve Pass@1 correctness code rates.

# **Chapter 4: Conclusion.**

The emergence and rapid advancement of large language models have recently imposed an enormous effect on various aspects and spheres of human life. These powerful models, trained on massive amounts of text data, have demonstrated remarkable capabilities in understanding, generating, and manipulating human language, thus pushing the boundaries of what is possible with AI, enabling breakthroughs in various applications such as machine translation, text summarization, question answering, and content generation. The impact of LLMs extends far beyond the realm of academia and research. Industries across the board, from healthcare and finance to entertainment and e-commerce, have recognized the potential of LLMs to transform their operations and enhance customer experiences.

Large language models will continue getting more efficient, ubiquitous and wholesome with time, representing more than ever necessity of protecting them from the adversarial users that aim at applying these models in socially malicious and harmful purposes. The aim of this study was to analyze the existing LLMs protection methods via theoretical and empirical analyses of existing and newly proposed methods in the conditions of machine-generated programming code detection. In order to achieve this aim, we conducted investigation of existing literature on this topic, introduced a complete examination of language & watermarking models’ fundamentals from the theoretical point of view and analyzed the performance metrics of WLLM and SWEET watermarking methods in the real-world conditions.

The aim of this research was successfully achieved. We defined the notions of a language model and a watermarking procedure, indicating how it is typically performed and emphasizing the properties of an “ideal” watermark, which include minimalized text distortion, independence from LLMs, no requirement on language model retraining and ability to defend from different kinds of adversarial attacks (resilience). Subsequently, we scrutinized the theoretical principles of the best of the existing code watermarking methods, called WLLM (“A Watermark for Large Language Models”) by Kirchenbauer et al. (2023a) and SWEET (Selective WatErmarking via Entropy Thresholding) by Lee et al. (2023). We showed that both models have essentially critical disadvantages that disallow them from being used in practice. After that, we defined three novel watermarking proposals, one of which aims at reinforcing the performance of WLLM and SWEET, and other two purposing at introducing the watermarking schemes that would not detrimentally affect the correctness of the generated code.

In experimental section we partially confirmed the imperfect nature of WLLM and SWEET, showing that implementation of moderate and strong versions of these watermarks into the code generation process turn out to negatively affect the correctness and efficiency of the code generated, while the application of a weak WLLM or SWEET occurs to be practically useless due to enormously low detectability rates. Thus, this subchapter outlined the main disadvantage of both models: WLLM and SWEET impose a detectability – efficiency trade-off on the Python code generation. Though SWEET is less exposed to this issue, concurrently showing better results compared to WLLM, it still suffers from its’ non-robust nature that makes this watermarking scheme vulnerable to various kinds of attacks, especially the paraphrasing ones.

This research may find its’ purpose in both theoretical and practical directions. One of the main theoretical aims of this paper was to popularize the topic of finding the best detection methods of LLM-generated code snippets, thus making this stream of literature more acknowledged. Moreover, the newly proposed three watermarking methods may be further investigated in the purposes of fully uncovering their potential, which may turn out to be quite useful in the context of real-world application.

Inevitably, this research also has certain limitations. From the theoretical side of analysis, our research could be supplemented with more details on the newly proposed watermarking schemes, including introduction of a larger number of possible code transformations that are the basis of Methods 1 and 3, provision of some statistical tests to check the presence of watermark, and the examination of the possibility to use these methods simultaneously or in the pair with other watermarking schemes. From the empirical side of analysis, future researchers may aim at checking the detectability rates of new methods and comparing them to performance of other watermarking or even post-hoc models. Moreover, some kind of resilience analysis may be provided in order to stress-test watermarking procedures and check their robustness to real-world adversarial influence in the form of malicious attacks.

Concluding, this research provides a comprehensive analysis of existing methods to protect LLM-generated output (namely, programming code) via delving into the history of post-hoc and text/code watermarking methods, examination of best watermarking schemes’ performance rates and introduction of some novel ways of watermarking Python code snippets. Still, the research that will succeed in alleviating or circumventing all of the limitations mentioned above, will be capable of exerting huge influence on the task of protecting low-entropy LLM-generated output, which is a challenging though an essential issue in the contemporary world. Amelioration of our results is the basis for future research.

# **List of References.**

1. Aaronson, S. (2023). ‘Reform’ AI Alignment with Scott Aaronson. *AXRP - the AI X-risk Research Podcast.* URL: https://axrp.net/episode/2023/04/11/episode-20-reform-ai-alignment-scott-aaronson.html
2. Atallah, M. & Raskin, V. (2001). Natural language watermarking: Design, analysis, and a proofof-concept implementation. In *Information Hiding*, pp. 185–200. *Springer, Springer Berlin Heidelberg.*
3. Austin, J., Odena, A. & Nye, M. (2021). Program synthesis with large language models. *arXiv preprint:2108.07732*. https://doi.org/10.48550/arXiv.2108.07732
4. Bafatakis, N. & Boecker, N. (2019). Python Coding Style Compliance on Stack Overflow. *Conference: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). Volume: 16*. DOI: 10.1109/MSR.2019.00042
5. Baldassini, F., et.al. (2024). Cross-Attention Watermarking of Large Language Models. *ICASSP 2024.* https://doi.org/10.1109/ICASSP48485.2024.10446397
6. Bhat, M. M. & Parthasarathy, S. (2020). How Effectively Can Machines Defend Against Machine-Generated Fake News? An Empirical Study. In *Proceedings of the First Workshop on Insights from Negative Results in NLP*, pp. 48–53. doi: 10.18653/v1/2020.insights-1.7.
7. Brassil, J. & Low, S. (1995). Electronic marking and identification techniques to discourage document copying. *IEEE Journal on Selected Areas in Communications, 13(8):1495–1504*. DOI: 10.1109/49.464718
8. Chen, M. & Tworek, J. (2021). Evaluating large language models trained on code. *arXiv preprint: 2107.03374.* https://doi.org/10.48550/arXiv.2107.03374
9. Chowdhery, A., et.al. (2023). PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research 24, 1-113.*
10. Christ, M., Gunn, S. & Zamir, O. (2023). Undetectable watermarks for language models*. arXiv preprint*: 2306.09194. https://doi.org/10.48550/arXiv.2306.09194
11. Devlin, J. & Chang, M. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *North American Chapter of the Association of Computer Logistics*. https://doi.org/10.48550/arXiv.1810.04805
12. Foltynek, T., Meuschke, N. & Gipp, B. (2019). Academic Plagiarism Detection: A Systematic Literature Review. *ACM Computing Surveys, 52(6)*: 112:1–112:42. https://doi.org/10.1145/3345317
13. Guo, B. & Zhang, X. (2023). How close is chatgpt to human experts? comparison corpus, evaluation, and detection. *arXiv preprint*:2301.07597. https://doi.org/10.48550/arXiv.2301.07597
14. Hazell, J. (2023). Large language models can be used to effectively scale spear phishing campaigns. *arXiv preprint*: 2305.06972.
15. He, X., Xu, Q., Lyu, L., Wu, F. & Wang, C. (2021). Protecting intellectual property of language generation apis with lexical watermark. In *AAAI Conference on Artificial Intelligence*. https://doi.org/10.48550/arXiv.2112.02701
16. He, X., Xu, Q., Zeng, Y., Lyu, L., Wu, F., Li, J. & Jia, R. (2022). Cater: Intellectual property protection on text generation apis via conditional watermarks. *In Advances in Neural Information Processing Systems*. https://doi.org/10.48550/arXiv.2209.08773
17. Kirchenbauer, J., Geiping, J., Wen, Y., Katz, J., Miers, I. & Goldstein, T. (2023). A watermark for large language models. *Proceedings of Machine Learning Research* *202*, pp. 17061-17084. https://doi.org/10.48550/arXiv.2301.10226
18. Kuditipudi, R., Thickstun J. & Hashimoto T. (2023). Robust Distortion-free Watermarks for Language Models. *TMLR*. https://doi.org/10.48550/arXiv.2307.15593
19. Lee, T. & Hong, S. (2023). Who Wrote this Code? Watermarking for Code Generation. *ACL ARR 2024 February Blind Submission*. https://doi.org/10.48550/arXiv.2305.15060
20. Liu, Y., Han, T., Ma, S., Zhang, J., Yang, Y., Tian, J., He, H., Li, A., He, M. & Liu, Z. (2023). Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-Radiology*, 100017. https://doi.org/10.1016/j.metrad.2023.100017
21. Mirsky, Y., Demontis, A. & Kotak, J. (2023). The threat of offensive AI to organizations. *Computers & Security, 124:103006.* https://doi.org/10.48550/arXiv.2106.15764
22. Mitchell, E., Lee, Y., Khazatsky, A., Manning, C. & Finn, C. (2023). Detectgpt: Zeroshot machine-generated text detection using probability curvature. *Proceedings of the 40th International Conference on Machine Learning, 1038,* pp. 24950–24962. https://doi.org/10.48550/arXiv.2301.11305
23. Palmer, A. (2023). People are using A.I. chatbots to write Amazon reviews. URL: https://www.cnbc.com/2023/04/25/
24. Radford, A. & Narasimhan, K. (2018). Improving Language Understanding by Generative Pre-Training. *Computer Science, Linguistics.*
25. Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C. & Sutskever, I. (2022). Robust Speech Recognition via A Watermark for Large Language Models. Large-Scale Weak Supervision. *arXiv preprint arXiv*: 2212.04356. doi: 10.48550/arXiv.2212.04356.
26. Sadasivan, V., Kumar, A. & Balasubramanian, S. (2023). Can ai-generated text be reliably detected*? ICLR 2024. arXiv preprint:2303.11156.* https://doi.org/10.48550/arXiv.2303.11156
27. Sandoval, G. & Pearce, A., et.al. (2023). Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. https://doi.org/10.48550/arXiv.2208.09727
28. Schulman, J., Zoph, B., Kim, C., Hilton, J., Menick, J., Weng, J., Uribe, J. F. C., Fedus, L., Metz, L., Pokorny, M., Gontijo-Lopes, R., Zhao, S., Vijayvergiya, A., Sigler, E., Perelman, A., Voss, C., Heaton, M., Parish, J., Cummings, D., Nayak, R., Balcom, V., Schnurr, D., Kaftan, T., Hallacy, C., Turley, N., Deutsch, N., Goel, V., Ward, J., Konstantinidis, A., Zaremba, W., Ouyang, L., Bogndonoff, L., Gross, J., Medina, D., Yoo, S., Lee, T., Lowe, R., Mossing, D., Huizinga, J., Jiang, R., Wainwright, C., Almeida, D., Lin, S., Zhang, M., Xiao, K., Slama, K., Bills, S., Gray, A., Leike, J., Pachocki, J., Tillet, P., Jain, S., Brockman, G. & Ryder, N. (2022). ChatGPT: Optimizing Language Models for Dialogue. URL: https://openai.com/blog/chatgpt/.
29. Suresh, T. & Ugare S. (2024). Is Watermarking LLM-Generated Code Robust? *arxiv preprint.* https://doi.org/10.48550/arXiv.2403.17983
30. Tian, E. (2023). Gptzero update v1. URL: https://gptzero.substack.com/ p/gptzero-update-v1
31. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P. & Bhosale, S. (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv*: 2307.09288. https://doi.org/10.48550/arXiv.2307.09288
32. Venugopal, A. & Uszkoreit, J. (2011). Watermarking the Outputs of Structured Prediction with an application in Statistical Machine Translation. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pp. 1363–1372. *Association for Computational Linguistics*. URL: https://aclanthology.org/D11-1126.
33. Wang, L., Yang, W. & Chen, D. (2023). Towards codable text watermarking for large language models. *ICLR 2024. arXiv preprint*:*2307.15992*. https://doi.org/10.48550/arXiv.2307.15992
34. Wu, K. & Pang, L. (2023). LLMDet: A Third Party Large Language Models Generated Text Detection Tool. *EMNLP 2023*. https://doi.org/10.48550/arXiv.2305.15004
35. Yang, X. & Cheng, W. (2023). Dna-gpt: Divergent n-gram analysis for training-free detection of gptgenerated text. *arXiv preprint*:2305.17359. https://doi.org/10.48550/arXiv.2305.17359
36. Zhang S., Roller S. & Goyal N. (2022). OPT: Open Pretrained Transformer Language Models. *. arXiv preprint.* https://doi.org/10.48550/arXiv.2205.01068
37. Zhang, J., Gu, Z., Jang, J., Wu, H., Stoecklin, M. P., Huang, H. & Molloy, I. (2018). Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security,* pp. 159–172*.* https://doi.org/10.1145/3196494.3196550
38. Zhao, W., Zhou, K., Li, J., Tang, T., Wang, X. & Hou, Y. (2023). A survey of large language models. *arXiv preprint*: 2303.18223. https://doi.org/10.48550/arXiv.2303.18223
39. Zhao, X. & Ananth, P. (2023). Provable robust watermarking for ai-generated text. *ICLR 2024*. https://doi.org/10.48550/arXiv.2306.17439
40. Zhao, X., Wang, Y. & Li, L. (2023). Protecting Language Generation Models via Invisible Watermarking. *Proceedings of the 40th International Conference on Machine Learning, 1774,* pp. 42187 – 42199. https://doi.org/10.48550/arXiv.2302.03162

# **Appendix.**

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Автоматически созданное описание

*Figure 1A. Example of HumanEval entry.*

1. https://scholar.google.com/ [↑](#footnote-ref-1)
2. https://blog.allenai.org/a-guide-to-language-model-sampling-in-allennlp-3b1239274bc3 [↑](#footnote-ref-2)
3. https://github.com/hhatto/autopep8 [↑](#footnote-ref-3)
4. https://github.com/openai/human-eval/tree/master [↑](#footnote-ref-4)
5. https://github.com/google-research/google-research/tree/master/mbpp [↑](#footnote-ref-5)
6. https://github.com/bigcode-project/starcoder [↑](#footnote-ref-6)