CODEIP: A Grammar-Guided Multi-Bit Watermark for Large Language Models of Code

Batu Guan[1] **Yao Wan**[1*] **Zhangqian Bi**[1] **Zheng Wang**[2] **Hongyu Zhang**[3]
Pan Zhou[1] **Lichao Sun**[4]
[1]Huazhong University of Science and Technology [2]University of Leeds
[3]Chongqing University [4]Lehigh University
{batuguan,wanyao,zqbi,panzhou}@hust.edu.cn, z.wang5@leeds.ac.uk
hyzhang@cqu.edu.cn, lis221@lehigh.edu

Abstract

Large Language Models (LLMs) have achieved remarkable progress in code generation. It now becomes crucial to identify whether the code is AI-generated and to determine the specific model used, particularly for purposes such as protecting Intellectual Property (IP) in industry and preventing cheating in programming exercises. To this end, several attempts have been made to insert watermarks into machine-generated code. However, existing approaches are limited to inserting only a single bit of information. In this paper, we introduce CODEIP, a novel multi-bit watermarking technique that inserts additional information to preserve crucial provenance details, such as the vendor ID of an LLM, thereby safeguarding the IPs of LLMs in code generation. Furthermore, to ensure the syntactical correctness of the generated code, we propose constraining the sampling process for predicting the next token by training a type predictor. Experiments conducted on a real-world dataset across five programming languages demonstrate the effectiveness of CODEIP in watermarking LLMs for code generation while maintaining the syntactical correctness of code.

1 Introduction

Large Language Models (LLMs), particularly those pre-trained on code, such as CodeGen (Nijkamp et al., 2022), Code Llama (Roziere et al., 2023), and StarCoder (Li et al., 2023a), have demonstrated great potential in automating software development. Notably, tools leveraging these LLMs, such as GitHub Copilot (Friedman, 2021), Amazon's CodeWhisperer (Amazon, 2023), and ChatGPT (OpenAI, 2023), are transforming how developers approach programming by automatically generating code based on natural language instructions and the context provided by existing code.

Although LLMs have shown significant potential in code generation, they also present challenges regarding the protection of Intellectual Property (IP) related to model architectures, weights, and training data, given the substantial costs associated with training a successful LLM (Li, 2024). Furthermore, there are also increasing concerns about the use of generative AI in programming courses (Bozkurt et al., 2023). A crucial method for safeguarding the IPs of LLMs and detecting programming misconduct is to determine if a specific piece of code is generated by a particular LLM.

Watermarking (Kirchenbauer et al., 2023) offers a potential solution to determine the origin of machine-generated content. This technique is shown to be effective in Computer Vision (CV) and Natural Language Processing (NLP) domains. It works by inserting information into multimedia formats (e.g., images and videos) without perceptibly diminishing the utility of content. By incorporating fingerprints such as owner/user ID, it supports leakage tracing, ownership identification, meta-data binding, and fortifying against tampering (Mohanty, 1999).

Existing watermarking techniques for LMs can be categorized into two groups: hard and soft watermarks. A hard watermark is typically inserted by utilizing a masked language model like BERT (Devlin et al., 2019) and RoBERTa (Liu et al., 2019) to replace tokens in generated content with synonyms. However, a hard watermark exhibits consistent patterns for different model inputs, compromising the protection strength. In contrast, soft watermarks are inserted during content generation, typically via manipulating the sampling probability distribution over the vocabulary during the decoding process of LLMs (Kirchenbauer et al., 2023). As soft watermarks can adapt to the generated content, they change across model outputs, improving the diversity and strength of watermarks.

Recently, several attempts have been made to-

*Corresponding Author.

Findings of the Association for Computational Linguistics: EMNLP 2024, pages 9243–9258

Figure 1: CODEIP can seamlessly embed multi-bit messages into LLMs while preserving the utility of the underlying code. "718084" is the ASCII value for "GPT".

wards watermarking LLMs for code generation, predominantly centered on two distinct approaches: generating a one-bit watermark to discern the machine-generated nature of the code (Lee et al., 2023) or embedding a hard watermark through a semantic-equivalent transformation of the generated code (Li et al., 2023b; Sun et al., 2023). We argue that a single-bit watermark carries little information and is inadequate to preserve enough provenance information like the vendor ID of an LLM. Moreover, the implementation of a hard watermark does not offer robust protection (Wang et al., 2024), as the easily detectable nature of the hard-coded watermarking patterns undermines its effectiveness.

To address the aforementioned limitations, this paper presents CODEIP, a grammar-guided *multi-bit soft watermarking* method for LLM-based code generation. CODEIP inserts a watermark message based on the probability logits of LLMs during the code generation process, thereby embedding a multi-bit message in the generated code. Moreover, CODEIP incorporates grammar information into the process of generating watermarked code, maximizing the likelihood of generating semantically correct code. This is achieved by training a type predictor to predict the subsequent grammar type of the next token, thereby enhancing the semantic correctness of the generated code.

Figure 1 illustrates the advantages of type predictor introduced in CODEIP. In this example, our objective is to insert the multi-bit message (i.e., model name) "718084" (corresponding to the ASCII value of "GPT") into its generated code. Without grammar guidance, the LLM inaccurately predicts the next token as ":". However, the grammar analysis indicates that the succeeding token is expected to be a keyword. Our CODEIP, which incorporates grammar constraints into the logit of LLMs, consistently tends to predict the correct token "in". This capability preserves the semantic correctness of the code during the insertion of watermarks into LLMs.

We assess the performance of CODEIP by inserting watermarks into code generated by three LLMs across five programming languages, namely Java, Python, Go, JavaScript, and PHP. Experimental results validate the efficacy of CODEIP, demonstrating an average watermark extraction rate of 0.95. Importantly, our method preserves the utility of the generated code, achieving 50% less CodeBLEU degradation compared to a baseline model without grammar constraints.

This paper makes the following contributions.

- It is the first to study the problem of embedding the soft multi-bit watermarks into code LLMs during the code generation process.
- It presents a new method that utilizes the grammatical constraints of programming languages to guide the manipulation of probability logits in LLMs, thereby preserving the utility of watermarked code.

Data Availability. All experimental data and source code used in this paper are available at https://github.com/CGCL-codes/naturalcc/tree/main/examples/codeip (Wan et al., 2022).

2 Preliminary

2.1 Code Generation

LLM-based code generation produces source code from high-level specifications or prompts. Typically, these specifications (prompts) are conveyed through natural language descriptions, supplemented by partial code elements such as function annotations and declarations, which are provided by users. Formally, let ρ denote a prompt, which can be tokenized into a sequence of tokens $\{w_0, w_1, \ldots, w_{|\rho|}\}$, where $|\cdot|$ denotes the length of a sequence. Let $\mathcal{V}$ denote the vocabulary used for mapping each token to corresponding indexes.

$$p_{\text{LM}}(w_i) = \text{softmax}\left(\mathcal{L}_{\text{LM}}(w_i|\rho, w_{0:i-1})\right) . \quad (1)$$

Here, $p_{\text{LM}}(w_i)$ denotes the probability distribution over the entire vocabulary $\mathcal{V}$, generated by the LM. We call the unnormalized score for each token in $\mathcal{V}$ produced by the LM as *model logit*. In this paper, the LM will always be an autoregressive Transformer (Vaswani et al., 2017) pre-trained on

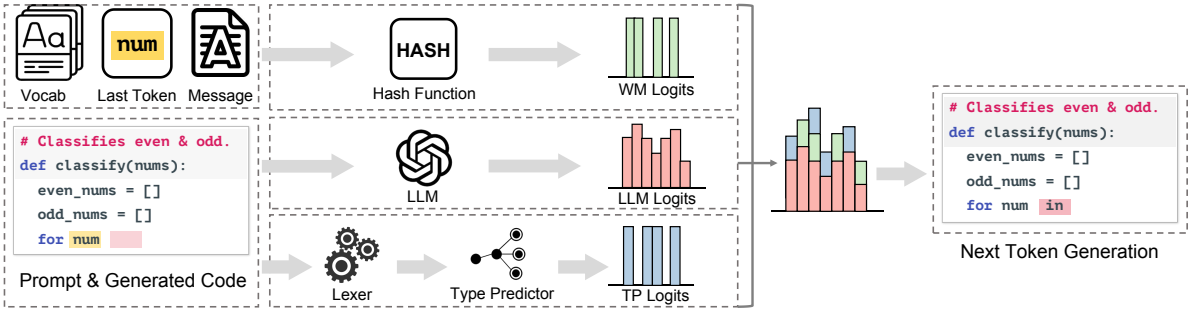


Figure 2: An overview of the watermark insertion process of CODEIP. The last token of the generated code and the message are used to compute watermark logits (WM Logits) by a hash function. The prompt and the whole generated code are firstly used by LLM to compute LLM logits and consequently input into a lexer and a type predictor to compute type predictor logits (TP Logits). The three are added together to compute the final logits, which are used for decoding in the next token generation stage.

source code, akin to the models in the GPT family, including Code Llama (Roziere et al., 2023) and StarCoder (Li et al., 2023a). Following this, the subsequent token w_i is sampled from $p_{\text{LM}}(w_i)$ using specific sampling strategies, such as greedy sampling (Berger et al., 1996) or multinomial sampling (Bengio et al., 2000). In this paper, we adopt the greedy sampling strategy (cf. Appendix E). Therefore, the next token will be sampled based on the following equation: $w_i = \arg\max_{w \in \mathcal{V}} p_{\text{LM}}(w)$.

2.2 The Problem: Watermarking the Code

In this paper, our goal is to insert a multi-bit watermark message into a code snippet during the generation process of LLMs. Typically, the watermarking algorithm comprises two stages: the insertion stage and the extraction stage.

During the process of inserting a watermark into the generated code, the initial consideration involves determining the specific message m to be inserted as the watermark. In practice, the model provider of an LLM can formulate a message, e.g. owner ID, to safeguard its model copyright. It is noteworthy that while the initial content of message m may encompass any characters, it undergoes conversion into a unique number before insertion. Specifically, given the prompt ρ and a watermark message m as inputs, the INSERT module produces a watermarked code $C = \text{INSERT}(\rho, m)$.

During the extraction stage, given an input snippet of code C, we expect that the module EXTRACT will produce its predicted watermark message $m' = \text{EXTRACT}(C)$.

In the context of this formulation, the primary objectives of our watermarking for LLMs of code are twofold: 1) to accurately insert the intent message as a watermark, and 2) to preserve the utility of the code without loss of semantics.

3 CODEIP

This section provides a detailed description of CODEIP. The CODEIP comprises two distinct stages, namely insertion and extraction. Initially, leveraging the decoding mechanism of existing LLMs, we use $\mathcal{L}_{\text{LM}}$ to denote the likelihood of each token in the vocabulary $\mathcal{V}$ to be inferred by the LLM. Subsequently, during the watermark insertion stage (cf. Sec. 3.1), we incorporate the watermark message by calculating a logit value $\mathcal{L}_{\text{WM}}$ to influence the prediction choice of tokens. Moreover, we present a novel application of context-free grammar and introduce another logit at the insertion stage (denoted as $\mathcal{L}_{\text{TP}}$), which signifies the probability associated with the grammatical type of the subsequent token, to guide token generation from the perspective of grammar (cf. Sec. 3.2). Finally, we integrate all the logits together (cf. Sec. 3.3) and explain the watermark extraction techniques (cf. Sec. 3.4).

3.1 Watermark Insertion

The watermark insertion architecture is depicted in Figure 2. Following Kirchenbauer et al. (2023), we insert the watermark into the generated code by modifying the probability distribution over the entire vocabulary $\mathcal{V}$ when LLM generates the next token. We first select a set of tokens from the vocabulary using a hash function. Based on the selected tokens, we compute the watermark logits, representing the likelihood of embedding the watermark message within each respective token.

Vocabulary Selection. The insight of inserting watermarks into code lies in selecting a set of tokens in the vocabulary under the control of the watermark message and enhancing their possibility of being generated during LLM decoding. We employ a hash function $\mathcal{H}$ to select tokens from the vocabulary $\mathcal{V}$. Specifically, assuming that LLM is generating the i-th token and the previous generation is denoted as $[w_0, w_1, \cdots, w_{i-1}]$, with watermark message represented by m. For any given token w in $\mathcal{V}$, the hash function will take (w, m, w_{i-1}) as input and map it to either 0 or 1. We consider tokens w that satisfy $\mathcal{H}(w, m, w_{i-1}) = 1$ as selected tokens, and our objective is to enhance their likelihood of being chosen by the LLM.

Watermark Logit. To augment the likelihood of generation, we calculate an additional logit referred to as the *watermark logit* $\mathcal{L}_{\text{WM}}$ and incorporate it into the existing model logit $\mathcal{L}_{\text{LM}}$. The implementation of the watermark logit $\mathcal{L}_{\text{WM}}$ relies on the outcomes of vocabulary partitioning. Assuming that the current LLM generates the i-th token w_i, preceded by the last token w_{i-1}, and denoting the watermark information as m, the watermark logit is computed as follows:

$$\mathcal{L}_{\text{WM}} = \mathbb{I}\left(\mathcal{H}(w, m, w_{i-1}) = 1\right) . \qquad (2)$$

Here, $\mathbb{I}$ denotes the indicator function. By assigning a value of 1 to $\mathcal{L}_{\text{WM}}$ for those selected tokens whose resultant computation via the hash function equals 1, we can effectively enhance the likelihood of such tokens being preferentially chosen during the decoding stage of LLM.

3.2 Grammar-Guided Watermarking

Conventional watermarking methods, which randomly insert a message by perturbing the generation process for each token, often result in the disruption of the semantics within the generated code. We posit that the generated code ought to adhere to the grammatical rules of the programming language. Consequently, we propose the integration of grammar constraints as a guiding principle in the code generation process. This inclusion is envisioned to maintain the utility of watermarked generated code.

Context-Free Grammar (CFG). A CFG serves as a formal system for describing the syntax of programming languages, and possesses sufficient expressiveness to represent the syntax of most programming languages (Hoe et al., 1986). Typically,

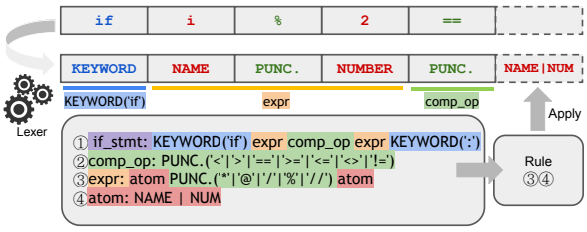


Figure 3: An example to highlight the role of CFG in ensuring the semantic correctness of generated code.

for a code snippet, a lexer, e.g. ANTLR (Parr and Quong, 1995), can transform it into a sequence of lexical tokens. Subsequently, under the constraints of CFG rules, we can infer the potential type of the subsequent lexical token. For instance, as illustrated in Figure 3, after transforming the original code "`if i % 2 ==`" into the sequence of lexical tokens, we can use CFG to infer the potential type of the subsequent lexical token as either "`NAME`" or "`NUM`", which could be helpful in the scenario of code generation.

Nonetheless, despite the constraints that CFG imposes on code, its direct application to the field of code generation still presents certain challenges. As demonstrated in the example of Figure 3, a CFG is capable of analyzing potential types for the subsequent lexical token. However, when multiple token types are considered valid next tokens, the utility of CFG in aiding code generation tasks becomes significantly limited, as it cannot calculate the probability distribution among these possible token types. Therefore, we train a lexical token-type predictor and intend to use it as a substitute for the CFG.

Lexical Token Type Predictor. We train a neural network to predict the lexical type of the next token. In particular, given the prompt and previously generated tokens, we initially employ a lexer to transform the given data into a sequence of lexical token types. Subsequently, this sequence is inputted into the predictor. The predictor then predicts a token type that will be outputted as the most probable lexical token type for the subsequent token.

In the context of LLM-based code generation, let ρ denote the prompt and G represent the generated code, assuming the LLM is currently generating the i-th token. For any given code snippet denoted as $S = [\rho; G_{0:i-1}]$, where $[\cdot;\cdot]$ denotes the concatenation of two elements, it is feasible to extract its token sequence $T = \text{Lexer}(S) = [\tau_0, \tau_1, \ldots, \tau_{l-1}]$ via lexical analysis, where $\tau \in \mathcal{T}$ denotes the lex-

ical token type and l denotes the length of lexical token sequence.

Subsequently, an LSTM (Hochreiter and Schmidhuber, 1997) is adopted to serve as the type predictor and to predict the token type of the subsequent token by inputting the token sequence T, as follows:

$$\tau_l = \text{TP}(T) = \text{LSTM}(\tau_0, \tau_1, \ldots, \tau_{l-1}) . \quad (3)$$

Other neural networks, such as the Transformer (Vaswani et al., 2017) can also be applied and we leave the exploration of other neural networks as our future work.

Type Predictor Logit. In order to mitigate the negative impact of watermarking on code utility, it is imperative to leverage our type predictor during the watermark insertion process, which is also the LLM decoding period. This necessitates transforming the predictive outcomes of the type predictor into a form of logit that can be added onto model logits. We name the new logit as *type predictor logit*, which can also be represented as $\mathcal{L}_{\text{TP}}$.

The type predictor logits are scores of tokens within $\mathcal{V}$. Consequently, it becomes imperative to construct a dictionary in advance that associates each type of lexical token with potential LLM tokens corresponding to that particular type. For instance, the `KEYWORD` lexical token type encompasses LLM tokens such as "`def`", "`if`", and "`else`", while the `Punctuation` lexical token type incorporates LLM tokens including "`(`", "`)`", "`;`", "`*`", and so forth. We denote this dictionary by $\Phi : \mathcal{T} \mapsto \mathcal{V}$. Thus, $\mathcal{L}_{\text{TP}}$ can be calculated as:

$$\mathcal{L}_{\text{TP}} = \mathbb{I}(w_i \in \Phi(\tau_{l+1})) . \quad (4)$$

Finally, we can get into the process of generating the i-th token w_i.

3.3 Putting it All Together

Finally, the i-th token generated by the LLM can be formulated as follows:

$$w_i = \arg\max_{w \in \mathcal{V}} \{\text{softmax}(\mathcal{L}_{\text{LM}} + \beta \mathcal{L}_{\text{WM}} + \gamma \mathcal{L}_{\text{TP}})\} . \quad (5)$$

Herein, β and γ represent hyperparameters for watermark logit $\mathcal{L}_{\text{WM}}$ and type predictor logit $\mathcal{L}_{\text{TP}}$.

3.4 Watermark Extraction

In the watermark insertion stage, we employ $\mathcal{L}_{\text{WM}}$ to insert a watermark w into the output G. Our strategy for watermark extraction involves enumerating all possible instances of the message (denoted as m'), recreating the process of watermark insertion, and identifying the instance of w that maximizes $\mathcal{L}_{\text{WM}}$, as follows:

$$m_{\text{ext}} = \arg\max_{m'} \left\{ \sum_{i=1}^{L} \mathcal{L}_{\text{WM}} \left(w_i \mid m', w_{i-1} \right) \right\} , \quad (6)$$

where m_{ext} denotes the message extracted using Eq. 6 and L denotes the length of token sequence in G. The insight of the extraction stage is that we determine that the most likely message causing the appearance of this generated code is the watermark inserted within this text.

4 Experimental Setup

4.1 LLMs and Dataset

To validate the effectiveness of our CODEIP, we choose three prominent LLMs: Code Llama (Roziere et al., 2023), StarCoder (Li et al., 2023a), and DeepSeek Coder (Bi et al., 2024a) as our target models. We insert the watermark into the code generated by these selected models. Note that, these models exist in different versions, each characterized by varying model sizes. In our experiments, we choose to employ the 7B model size, limited by the computation resources. We select Java, Python, Go, JavaScript, and PHP from CodeSearchNet (Husain et al., 2019) dataset and use the docstrings and function declarations as prompts. For each prompt, the LLMs generate the next 200 tokens. Note that here we do not adopt HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) datasets as our evaluation datasets. This is because their code length is generally too short (cf. Appendix D) and not suitable for inserting watermarks. The relationship between the length of generated code and the extraction rate is studied in Sec. 5.3.

4.2 Implementation Details

For all three LLMs, we implement a temperature of 0.75, a repetition penalty of 1.2, and no repeat n-gram size of 10. Given the distinct training processes of various LLMs, we establish the parameters (β, γ) as $(5, 3)$ for Code Llama and StarCoder, $(6, 4)$ for DeepSeek Coder. We set the watermark message to be 2024 in our experiment and the whole possible watermark message set is $[0, 2^{20}]$, which means we can insert a 20-bit message at most. The type predictor is an LSTM model, which encompasses an embedding layer character-

LLM	Strategy	Java	Python	Go	JavaScript	PHP
Code Llama	w/ WM + w/o TP	0.90	0.93	0.87	0.98	0.97
	w/ WM + w/ TP	0.92	0.93	0.86	1.00	0.97
StarCoder	w/ WM + w/o TP	0.88	0.98	0.90	0.97	0.96
	w/ WM + w/ TP	0.86	0.97	0.87	0.96	0.96
Deepseek Coder	w/ WM + w/o TP	0.99	0.95	0.87	1.00	1.00
	w/ WM + w/ TP	0.99	1.00	0.91	1.00	1.00

Table 1: The results of watermark extraction rate for different models with different strategies, where "WM" denotes Watermark and "TP" denotes Type Predictor.

ized by an embedding dimensionality of 64. The hidden state dimensionality of the LSTM is 128. In our experiment, we train a type predictor for each language involved, given the distinct grammatical structures inherent to each language. We also compare the ability of CODEIP to preserve code quality with other methods, particularly the vanilla methods from Wang et al. (2024) and Yoo et al. (2024). These methods have similar time consumption for inserting a watermark into a code snippet, making them suitable for comparison with our approach. The parameters for these comparative experiments have been adjusted to align with the code generation scenario. All experiments are conducted on a Linux server with 128GB memory, with a single 32GB Tesla V100 GPU.

4.3 Evaluation Metrics

To evaluate the effectiveness of watermarking, one objective is to assess whether the watermark can be extracted from the generated code. Specifically, we select 100 functions from the dataset for each programming language and extract the docstrings and declarations of each function to serve as prompts for LLMs generation. We employ the extraction rate of watermarks as a metric to measure the efficacy of watermarking, reflecting the percentage of watermarks successfully extracted from the embedded code. Assuming there are N prompts in the dataset, these N prompts, when input to the LLM with CODEIP, will generate N segments of code with watermarks. By using CODEIP, watermarks in M segments of code are successfully extracted. The extraction rate will be calculated as follows.

$$\text{Extraction Rate} = \frac{M}{N}. \tag{7}$$

To validate the utility of watermarked code, we adopt the CodeBLEU (Ren et al., 2020) metric, which has been widely adopted in the evaluation of code generation. The CodeBLEU metric can be depicted as follows.

$$\begin{aligned}\text{CodeBLEU} = \eta \cdot \text{BLEU} + \lambda \cdot \text{BLEU}_{\text{weight}} \\ + \mu \cdot \text{Match}_{\text{ast}} + \xi \cdot \text{Match}_{\text{df}}.\end{aligned} \tag{8}$$

Here, BLEU is computed utilizing the conventional BLEU method as delineated by (Papineni et al., 2002). The term $\text{BLEU}_{\text{weight}}$ refers to a weighted n-gram match that is derived from juxtaposing hypothesis code and reference code tokens with varying weights. Furthermore, $\text{Match}_{\text{ast}}$ signifies a syntactic AST match which delves into the syntactic information inherent in the code. Lastly, Match_{df} denotes a semantic dataflow match that takes into account the semantic congruity between the hypothesis and its corresponding reference.

In our experiments, we adopt the parameters recommended by Ren et al. (2020) in their original paper, namely $(\eta, \lambda, \mu, \xi) = (0.10, 0.10, 0.40, 0.40)$. Note that, here we do not adopt the Pass@k metric (Chen et al., 2021), which has been widely adopted to evaluate the LLMs for code generation. This is because the test cases are missing in our used CodeSearchNet dataset. A detailed explanation of metrics is in Appendix C.

5 Results and Analysis

5.1 Extraction Rate of Watermarks

Table 1 shows a comparison among different kinds of watermarking strategies. Generally, under both watermarking strategies, the extraction rates consistently surpass 0.90 in most programming languages, indicating the efficacy of our watermarking techniques in the context of LLMs for code generation. Taking DeepSeek Coder as an example, our watermarking strategy, both with and without the type predictor ("w/ WM + w/o TP" and "w/ WM + w/ TP"), demonstrates an impressive extraction rate of 0.99 for Java and 1.00 for PHP. Moreover, the fact that the presence or absence of a type predictor has no obvious effect on the outcome is consistent

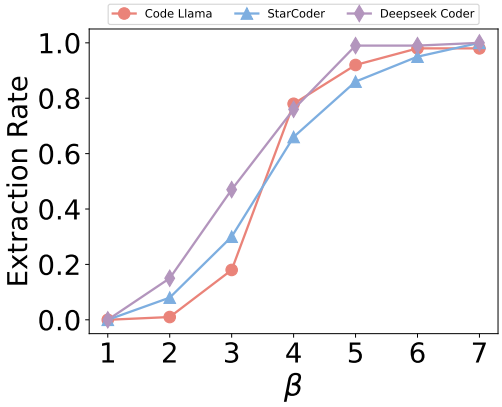


Figure 4: Impact of parameter β on the extraction rate of generated Java code.

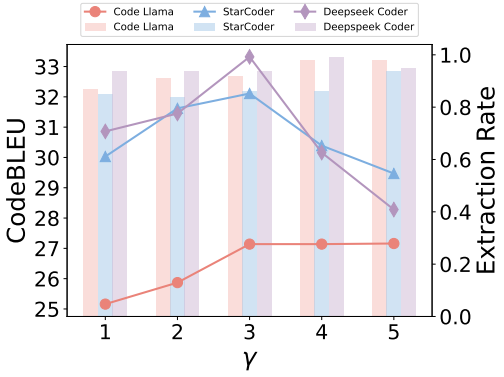


Figure 5: Impact of parameter γ on CodeBLEU score and extraction rate of generated Java code. Lines are for CodeBLEU and bars are for extraction rate.

with our initial expectations, as the type predictor is designed to prioritize the preservation of the utility of the generated code.

5.2 Watermark *vs* Code Quality

We further explore the impact of watermarking strategies on the utility of generated code. Table 2 illustrates the overall performance of different LLMs when paired with different logits, measured by CodeBLEU. From this table, it is evident that the use of watermark logit leads to a decrease in CodeBLEU scores for code generation across various models and languages, and with the subsequent incorporation of the type predictor logit, a distinct resurgence in CodeBLEU scores is observed across most settings. It should also be noticed that the performance of the logit predictor is superior to other similar works (Wang et al., 2024; Yoo et al., 2024). This emphasizes the significant efficacy of the type predictor in preserving the quality of code.

5.3 Parameter Analysis

The Impact of Parameter β. We conduct experiments on the variation in extraction rates when adjusting parameter β under three LLMs. We only show the result of Java as an example in this section and more results can be seen in Appendix F.1. In Figure 4, it can be seen that as β continues to increase, the extraction rate of watermarks is also constantly increasing. When β exceeds 5, an extraction rate of approximately 0.9 can essentially be achieved, which is relatively ideal. It indicates that watermark logit has a positive effect on whether watermarks can be extracted.

The Impact of Parameter γ. We conduct experiments on three LLMs by varying parameter γ, aiming to investigate the impact of γ on the CodeBLEU score and extraction rate of generated code. The experimental results of Java, as depicted in Figure 5, reveal a noteworthy trend. The initial augmentation of γ visibly improves the quality of the generated code. Nevertheless, as augmentation progresses beyond a certain threshold, a discernible decline in CodeBLEU becomes evident. One plausible explanation for this inconsistency may stem from the inherent contradiction in tokenization, namely, the disparity between prevalent tokenization methods utilized by LLMs (e.g., WordPiece (Schuster and Nakajima, 2012) and BPE (Sennrich et al., 2015)), and those employed by lexers.

For example, the LLM subtokens "`ran`" and "`ge`", when combined, can constitute the lexical token "`range`" which can be recognized during lexical analysis. Assuming the generated code to be "`for i in ran`", the subsequent LLM subtoken to be generated is most likely to be "`ge`", thereby rendering the generated code as "`for i in range`". However, from the perspective of a lexer, the token "`ran`" could potentially be classified as type "`NAME`", leading to the lexical token type being calculated as "`PUNCTUATION`", and thereby selecting "`:`". Hence, the generation of code will be transformed into "`for i in ran:`". This contradiction caused by different segmentation methods between LLM tokenizer and lexical analysis can also lead to performance degradation when γ is high.

Moreover, we measure the extraction rate under various γ settings and observe that changes in γ result in only minor fluctuations in the extraction rate. Thus, we conclude that the parameter γ primarily affects the utility of the generated code, with minimal impact on the extraction rate.

LLM	Strategy	Java	Python	Go	JavaScript	PHP
Code Llama	Unwatermarked	28.99	22.56	31.73	23.01	44.56
	Wang et al. (2024)	23.76 (-5.23)	10.04 (-12.52)	21.52 (-10.21)	16.32 (-6.69)	32.85 (-11.71)
	Yoo et al. (2024)	26.87 (-2.12)	8.91 (-13.65)	**28.29 (-3.44)**	12.79 (-10.22)	20.74 (-23.82)
	w/ WM + w/o TP	23.35 (-5.64)	12.04 (-10.52)	22.44 (-9.29)	16.47 (-6.54)	40.47 (-4.09)
	w/ WM + w/ TP	**27.14 (-1.85)**	**12.25 (-10.31)**	26.49 (-5.24)	**20.83 (-2.18)**	**40.61 (-3.95)**
StarCoder	Unwatermarked	39.16	17.74	27.61	24.06	42.60
	Wang et al. (2024)	27.29 (-11.87)	17.20 (-0.54)	14.84 (-12.77)	16.81 (-7.25)	36.44 (-6.16)
	Yoo et al. (2024)	22.34 (-16.82)	10.93 (-6.81)	**22.54 (-5.07)**	15.08 (-8.98)	17.36 (-25.24)
	w/ WM + w/o TP	25.70 (-13.46)	17.60 (-0.14)	13.39 (-14.22)	15.25 (-8.81)	40.11 (-2.49)
	w/ WM + w/ TP	**32.11 (-7.05)**	**18.16 (+0.42)**	17.55 (-10.06)	**19.18 (-4.88)**	**40.14 (-2.46)**
DeepSeek Coder	Unwatermarked	32.10	19.68	33.10	23.97	42.29
	Wang et al. (2024)	24.84 (-7.26)	15.95 (-3.73)	26.38 (-6.72)	19.61 (-4.36)	36.84 (-5.45)
	Yoo et al. (2024)	21.78 (-10.32)	17.62 (-2.06)	26.56 (-6.54)	17.18 (-6.79)	18.09 (-24.20)
	w/ WM + w/o TP	25.55 (-6.55)	**18.35 (-1.33)**	26.93 (-6.17)	17.88 (-6.09)	**43.40 (+1.11)**
	w/ WM + w/ TP	**31.22 (-0.88)**	13.57 (-6.11)	**29.32 (-3.78)**	**19.65 (-4.32)**	**43.40 (+1.11)**

Table 2: CodeBLEU scores for different models with different strategies. The value in () represents the disparity in quality (CodeBLEU) between watermarked and unwatermarked code.

The Impact of Generated Code Length. We also investigate the influence of generated code length, measured in terms of the number of tokens produced, on the effectiveness of watermark insertion. Our findings reveal a positive correlation between code length and the successful extraction rate, as depicted in Figure 6. This observation underscores that the successful extraction rate of our watermark remains contingent on the length of the generated code. Specifically, shorter lengths of generated code lead to diminished distinctions between watermarked and unwatermarked code, consequently presenting a heightened challenge in extracting watermarks within such code.

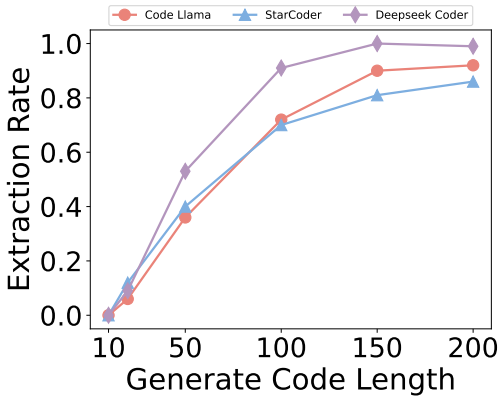


Figure 6: Impact of generated code length on the extraction rate of Java code.

5.4 Resistance to Crop Attack

To underscore the robustness of our watermarking strategies, we consider a hypothetical scenario where developers use only a portion, rather than the entire generated code, to undermine the watermark—a situation termed a "Crop Attack". This involves subjecting the generated code to crop rates of 0.25 and 0.5, representing the removal of 25% and 50% of the code, respectively. The results are presented in Table 3. Examination of the table reveals that, in most cases, our watermark's effectiveness only experiences a slight reduction under such rigorous attacks. These findings strongly indicate that our watermark exhibits notable resistance to crop attacks, demonstrating its robustness.

LLM	Rate	Java	Python	Go	JS	PHP
Code Llama	0	0.92	0.93	0.86	1.00	0.97
	0.25	0.89	0.95	0.75	0.96	0.94
	0.50	0.71	0.85	0.51	0.87	0.87
StarCoder	0	0.86	0.97	0.87	0.96	0.96
	0.25	0.81	0.95	0.85	0.93	0.95
	0.50	0.63	0.96	0.79	0.85	0.92
DeepSeek Coder	0	0.99	1.00	0.91	1.00	1.00
	0.25	0.98	0.99	0.77	0.94	0.95
	0.50	0.91	0.90	0.56	0.90	0.87

Table 3: The performance of CODEIP in code watermarking against crop attack.

6 Related Work

LLM-based Code Generation. The roots of code generation can be traced back several decades (Backus et al., 1957; Waldinger and Lee, 1969). Recently, many works focus on the intersection of deep learning and tasks of code (Wan et al.), such as code summarization (Wan et al., 2018; Alon et al., 2018), code search (Wan et al., 2020), code completion (Li et al., 2024; Sun et al., 2024b) and code generation (Bi et al., 2024b; Sun et al., 2024a). Currently, LLMs especially those pretrained on code, such as DeepSeek Coder (Bi et al.,

2024a), Code Llama (Roziere et al., 2023), CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023a), and CodeGeeX2 (Zheng et al., 2023), have emerged as dominant forces in code generation. Leveraging the capabilities of these LLMs, several commercial tools are reshaping the programming landscape for developers, including GPT-3.5 (OpenAI, 2023), Gemini (Google, 2024), and GitHub Copilot (Microsoft, 2024).

Software Watermarking. The software watermarking problem has been studied since 1996 by Davidson and Myhrvold (1996), who altered code blocks or operand order to insert watermarks. Qu and Potkonjak (1998) proposed a software watermark method based on graph coloring problem and graph structure of the code, which was further developed by Myles and Collberg (2004), Zhu and Thomborson (2006). These rule-based early methods are often constrained by the usage scenarios and various attack techniques.

Recently, several works (Yang et al., 2024; Li et al., 2023b) have been focusing on watermarking the code generated by LLMs. They utilized a post-processing approach, whereby watermarks are inserted through transformations applied to the code subsequent to its generation by the model. However, these techniques have several limitations, including their specificity to a single language and their vulnerability to counterfeiting once the watermarking method is disclosed, which restricts their applicability. Additionally, some multi-bit watermarking techniques (Wang et al., 2024; Yoo et al., 2024) have been proposed, but these approaches tend to reduce the utility of the generated text.

Machine Generated Text Identification. The task of identifying machine-generated text has always been of paramount importance. Early research focused on adding watermarks to arbitrary texts, while in recent years, studies on text watermarking have started their attempts to distinguish between machine-generated and human-generated texts. An intuitive approach is to treat it as a binary classification task, accomplished by training a model (Solaiman et al., 2019; Bakhtin et al., 2019). Another approach is to identify model-generated text by detecting features of the generated text. Tay et al. (2020) distinguished texts by detecting detectable artifacts in the generated text, such as sampling methods, top-k probabilities, etc. There is also a dataset created for testing models ability to distinguish machine-generated text from human-written text (Zhang et al., 2024). In 2023, Kirchenbauer et al. (2023) introduced a novel method for embedding watermarks into text during model inference by altering the selection probabilities of certain tokens. Lee et al. (2023) extended this method to code generation, incorporating threshold-controlled watermark inclusion.

7 Conclusion

In this paper, we propose CODEIP to watermark the LLMs for code generation, with the goal of safeguarding the IPs of LLMs. We insert watermarks into code generated by the model, and introduce grammatical information into the watermark generation process by designing a type predictor module to safeguard the utility of generated code. Comprehensive experimental findings affirm that CODEIP exhibits a notable extraction rate, excels in safeguarding code semantics, and demonstrates a degree of resilience against attacks. In our future work, we plan to persistently advance toward more secure LLM-powered software engineering through the continuation of our research.

8 Limitations

In our experiments, we adopt CodeBLEU for evaluation, which is a commonly used metric in assessing the quality of code generation. In our future work, we will employ additional evaluation metrics to assess the experimental results. We also strive to find datasets suitable for our work that can be evaluated using other metrics, such as Pass@k. Furthermore, the experiments have substantiated that our watermark exhibits a certain degree of robustness under crop attacks, as this is the most easily implemented attack method in model copyright scenarios. Other forms of attacks such as variable name obfuscation could potentially degrade the readability of generated code, thus making them less likely to be employed in attacks aimed at infringing model copyrights, which is an assault we aim to prevent. We will persistently investigate and enhance the robustness of our watermark to make it applicable for more protection scenarios.

Acknowledgements

This work is supported by the National Natural Science Foundation of China under grant No. 62102157, and the Major Program (JD) of Hubei Province (Grant No. 2023BAA024). We thank all the reviewers for their insightful comments.

References

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

C Amazon. 2023. Ai code generator—amazon codewhisperer.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, Lois M Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, Harold Stern, et al. 1957. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198.

Anton Bakhtin, Sam Gross, Myle Ott, Yuntian Deng, Marc'Aurelio Ranzato, and Arthur Szlam. 2019. Real or fake? learning to discriminate machine from human generated text. *arXiv preprint arXiv:1906.03351*.

Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A neural probabilistic language model. *Advances in neural information processing systems*, 13.

Adam L. Berger, Vincent J. Della Pietra, and Stephen A. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Comput. Linguist.*, 22(1):39–71.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024a. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024b. Iterative refinement of project-level code context for precise code generation with compiler feedback. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 2336–2353. Association for Computational Linguistics.

Aras Bozkurt, Xiao Junhong, Sarah Lambert, Angelica Pazurek, Helen Crompton, Suzan Koseoglu, Robert Farrow, Melissa Bond, Chrissi Nerantzi, Sarah Honeychurch, et al. 2023. Speculative futures on chatgpt and generative artificial intelligence (ai): A collective reflection from the educational landscape. *Asian Journal of Distance Education*, 18(1):53–130.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Robert I Davidson and Nathan Myhrvold. 1996. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL*, pages 4171–4186.

Nat Friedman. 2021. Introducing github copilot: your ai pair programmer. *URL https://github. blog/2021-06-29-introducing-github-copilot-ai-pair-programmer*.

Google. 2024. Gemini. https://deepmind.google/technologies/gemini/. [Online; accessed 1-Feb-2024].

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Alfred V Hoe, Ravi Sethi, and Jeffrey D Ullman. 1986. *Compilers—principles, techniques, and tools*. Pearson Addison Wesley Longman.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2023. A watermark for large language models. *arXiv preprint arXiv:2301.10226*.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. 2023. Who wrote this code? watermarking for code generation. *arXiv preprint arXiv:2305.15060*.

Bolun Li, Zhihong Sun, Tao Huang, Hongyu Zhang, Yao Wan, Ge Li, Zhi Jin, and Chen Lyu. 2024. Ircoco: Immediate rewards-guided deep reinforcement learning for code completion. *Proceedings of the ACM on Software Engineering*, 1(FSE):182–203.

Chuan Li. 2024. Demystifying gpt-3 language model: A technical overview. https://lambdalabs.com/blog/demystifying-gpt-3/. [Online; accessed 1-Feb-2024].

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Zongjie Li, Chaozheng Wang, Shuai Wang, and Cuiyun Gao. 2023b. Protecting intellectual property of large language model-based code generation apis via watermarks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2336–2350.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv:1907.11692*.

Microsoft. 2024. Microsoft Copilot. https://www.microsoft.com/zh-cn/microsoft-copilot. [Online; accessed 1-Feb-2024].

Saraju P Mohanty. 1999. Digital watermarking: A tutorial review. *URL: http://www. csee. usf. edu/˜ smohanty/research/Reports/WMSurvey1999Mohanty. pdf*.

Ginger Myles and Christian Collberg. 2004. Software watermarking through register allocation: Implementation, analysis, and attacks. In *Information Security and Cryptology-ICISC 2003: 6th International Conference, Seoul, Korea, November 27-28, 2003. Revised Papers 6*, pages 274–293. Springer.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2023. chatgpt. http://chat.openai.com. [Online; accessed 1-Feb-2023].

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Terence J. Parr and Russell W. Quong. 1995. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

Gang Qu and Miodrag Potkonjak. 1998. Analysis of watermarking techniques for graph coloring problem. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 190–193.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Mike Schuster and Kaisuke Nakajima. 2012. Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5149–5152. IEEE.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. 2019. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203*.

Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. 2023. Codemark: Imperceptible watermarking for code datasets against neural code completion models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1561–1572.

Zhihong Sun, Chen Lyu, Bolun Li, Yao Wan, Hongyu Zhang, Ge Li, and Zhi Jin. 2024a. Enhancing code generation performance of smaller models by distilling the reasoning ability of llms. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, pages 5878–5895. ELRA and ICCL.

Zhihong Sun, Yao Wan, Jia Li, Hongyu Zhang, Zhi Jin, Ge Li, and Chen Lyu. 2024b. Sifting through the chaff: On utilizing execution feedback for ranking the generated code candidates. *arXiv preprint arXiv:2408.13976*.

Yi Tay, Dara Bahri, Che Zheng, Clifford Brunk, Donald Metzler, and Andrew Tomkins. 2020. Reverse engineering configurations of neural text generation models. *arXiv preprint arXiv:2004.06201*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Richard J Waldinger and Richard CT Lee. 1969. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252.

Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Computing Surveys*.

Yao Wan, Yang He, Zhangqian Bi, Jianguo Zhang, Yulei Sui, Hongyu Zhang, Kazuma Hashimoto, Hai Jin, Guandong Xu, Caiming Xiong, et al. 2022. Naturalcc: an open-source toolkit for code intelligence. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 149–153.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2020. Multi-modal attention network learning for semantic source code retrieval. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 13–25. IEEE Press.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 397–407.

Lean Wang, Wenkai Yang, Deli Chen, Hao Zhou, Yankai Lin, Fandong Meng, Jie Zhou, and Xu Sun. 2024. Towards codable watermarking for injecting multi-bits information to llms. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Borui Yang, Wei Li, Liyao Xiang, and Bo Li. 2024. Srcmarker: Dual-channel source code watermarking via scalable code transformations. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4088–4106, Los Alamitos, CA, USA. IEEE Computer Society.

KiYoon Yoo, Wonhyuk Ahn, and Nojun Kwak. 2024. Advancing beyond identification: Multi-bit watermark for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 4031–4055. Association for Computational Linguistics.

Qihui Zhang, Chujie Gao, Dongping Chen, Yue Huang, Yixin Huang, Zhenyang Sun, Shilin Zhang, Weiye Li, Zhengyan Fu, Yao Wan, and Lichao Sun. 2024. LLM-as-a-coauthor: Can mixed human-written and machine-generated text be detected? In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 409–436, Mexico City, Mexico. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

William Zhu and Clark Thomborson. 2006. Recognition in software watermarking. In *Proceedings of the 4th ACM international workshop on Contents protection and security*, pages 29–36.

A Lexical Token Type

Despite the diversity in syntax among various programming languages, consistency remains at the lexical analysis level. That is, the types of tokens parsed out by lexical analysis are fundamentally similar. The text box below presents potential token types that may be parsed following lexical analysis.

```
'Token',      'Comment',      'Error',
'Escape',     'Generic',      'Keyword',
'Literal',    'Name',         'Operator',
'Other', 'Punctuation', 'Text'
```

B Learning the Type Predictor

Formally, the type predictor accomplishes the task of the next lexical token prediction. We adhere to conventional training methodologies for this particular task to train it. For a given programming language, we postulate that the collected code dataset of this particular language is denoted as $\mathcal{D}$, and each segment of code within this dataset as $d \in \mathcal{D}$. To facilitate the acquisition of pertinent language grammar by the type predictor, we initially employ a lexer specific to that language to transform each instance of d into a corresponding lexer token sequence. Taking into account that the possible token type of the subsequent word is typically associated with the types of nearby tokens, for predicting the type of the i-th token, we extract n preceding token types from the sequence to predict this i-th token type. Hence, for the dataset $\mathcal{D}$, our learning objective can be formulated as follows:

$$\mathcal{J}(\mathcal{D}) = \sum_{d \in \mathcal{D}} \sum_{i=n}^{|T_d|} \log p_{\text{LSTM}}(l_i | l_{(i-n):i}) \,, \qquad (9)$$

where $\mathcal{J}$ is the loss function utilized during the training of type predictor, and $|T_d|$ denotes the length of lexical token type sequence of original code d.

After training, each type predictor can achieve an accuracy rate of over 70% on the test set.

C Detailed Explanation of Metrics

In our experiments, we opt to assess the effectiveness of our watermarking system by detecting the extraction rate, rather than employing metrics such as FPR and AUROC. This is because, while one-bit watermark technology can only distinguish whether an article is generated by a machine, multi-bit watermarks can embed more information. Therefore, the ability to extract the information within is a more critical evaluation criterion.

In fact, based on the experimental results, we find that the false positive rate is 0, i.e., there is no record of successfully extracting watermarks from codes written by natural programmers. Theoretically, it is also not difficult to see that extracting the correct watermark from non-machine-generated text is very challenging. Assuming there are N possible watermark extraction results, for naturally generated text that is not machine-made, the results of watermark extraction can be considered to be uniformly distributed, hence the probability

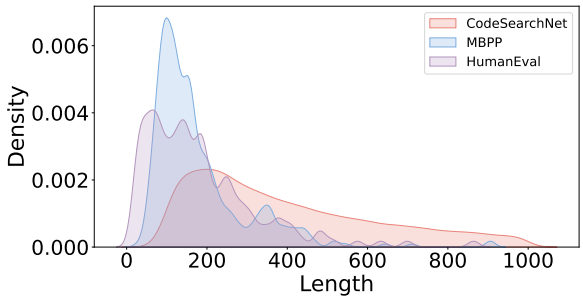


Figure 7: The length distribution of code in CodeSearchNet, MBPP, and HumanEval datasets. For better readability, code in CodeSearchNet exceeding 1000 characters has been truncated. The length is measured in characters.

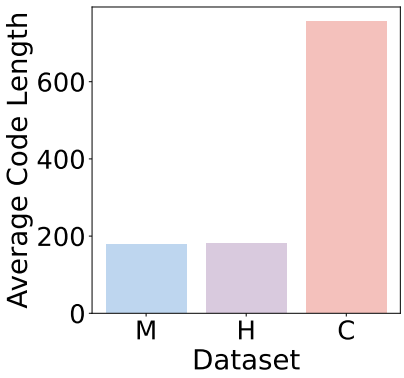


Figure 8: The average code length for CodeSearchNet, MBPP, and HumanEval, measured in characters. On the x axis, "M" represents MBPP, "H" represents HumanEval, and "C" represents CodeSearchNet.

of extracting the correct watermark is $1/N$. In experiments, the total number of possible extraction results is set to 2^{20}, under such circumstances, the probability of extracting the correct watermark is less than one in a million. Therefore, metrics such as FPR and AUROC are not suitable in our experiments.

D Dataset Analysis

In Figure 7, we examine the code lengths across three datasets: MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021), and CodeSearchNet (Husain et al., 2019). Analysis of both length distribution and average length reveals a notable distinction: the CodeSearchNet dataset exhibits significantly longer code lengths compared to the other two datasets.

E Sampling Strategy Selection

In our experiment we use greedy sampling as our decoding strategy. Other decoding strategies like beam search, top-p sampling, etc. can also be used. The selection of the decoding strategy will not affect our evaluation of CodeIP. Due to the convenience of greedy sampling, we opt for this strategy.

F More Results on Parameters

F.1 Results of Parameter β

We present additional results (Figure 9) demonstrating variations in extraction rate as β varies.

F.2 Results of Parameter γ

We present additional results regarding the variation of γ with the change in CodeBLEU score and extraction rate as shown in Figure 10 and Figure 11.

F.3 Results of Generated Code Length

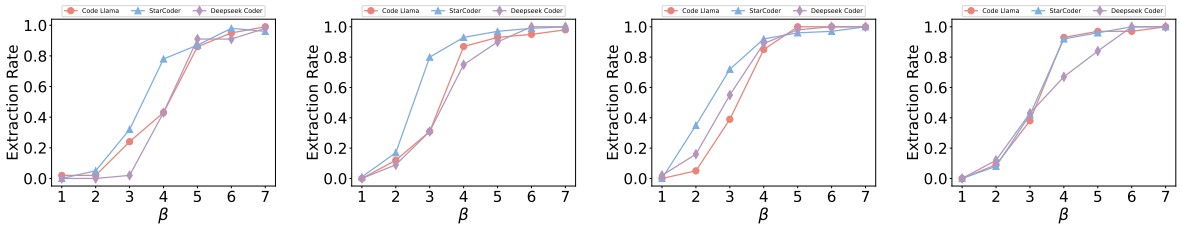
In Figure 12, we present additional results illustrating how the extraction rate varies with different values of generated code length.

G Case Study

In Figure 13, we demonstrate examples of the generation code of LLM under three different strategies(w/o WM + w/o TP, w/ WM + w/o TP, and w/ WM + w/ TP), and the watermark message is the number "1012". The prompt contains the docstring and declaration of the function.

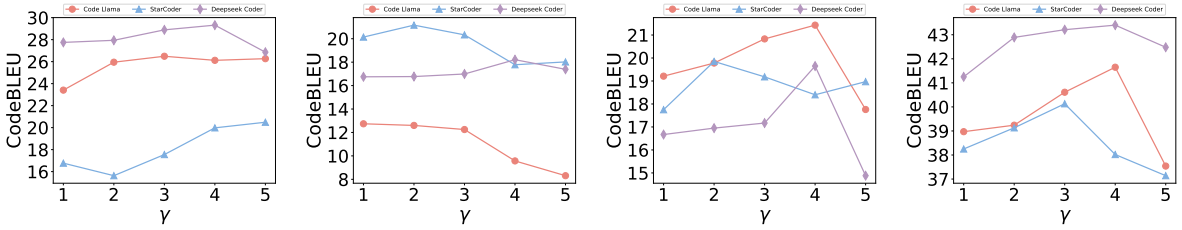
From Figure 13(a), we can see that when watermark logit and type predictor logit are not applied during the decoding stage of LLM, it generates some normal Python code, and in this scenario, no watermark is inserted in the code because watermark logit is not applied. When only the watermark logit is added to the model logit, the LLM starts to generate large sections of comments, which is meaningless to the implementation of the function. The reason is supposed to be that the watermark logits enhance the generation probability of comment symbols like "#" and "' ' '", who then affect the LLM to generate comments rather than codes, which do harmness to code utility. Subsequently, when type predictor logits are also added to the model logits, the generation code of LLM resumes to normal and generates complete code to implement the function shown in the prompt.

As illustrated in Figure 13(b), the LLM generated nearly identical outputs under the three strategies. In this particular instance, no conspicuous grammatical errors were detected, and the outputs of both strategies - w/ WM + w/o TP and w/ WM +



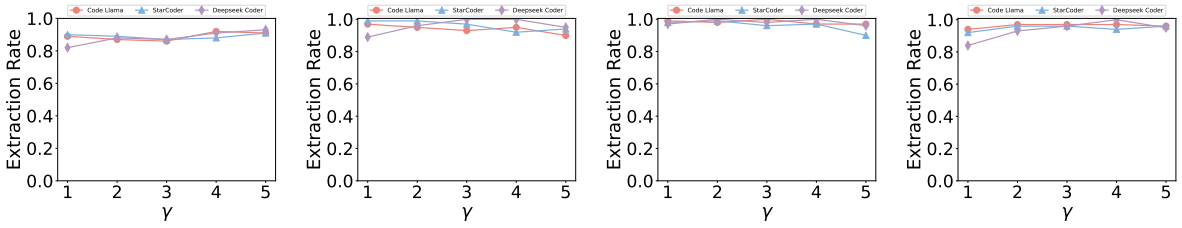
(a) Go (b) Python (c) JavaScript (d) PHP

Figure 9: Impact of parameter β on extraction rate of generated code.



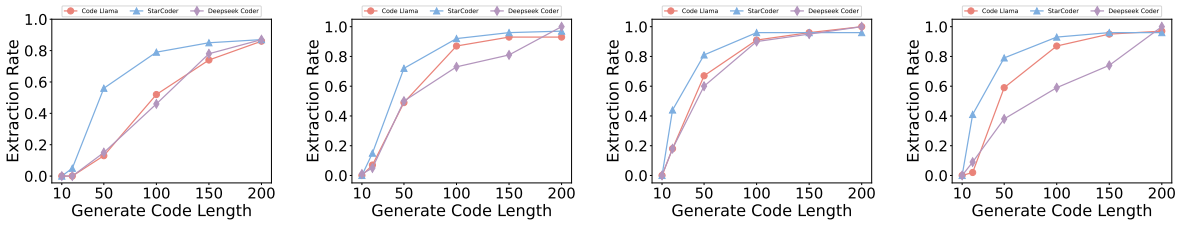
(a) Go (b) Python (c) JavaScript (d) PHP

Figure 10: Impact of parameter γ on CodeBLEU score of generated code.



(a) Go (b) Python (c) JavaScript (d) PHP

Figure 11: Impact of parameter γ on extraction rate of generated code.



(a) Go (b) Python (c) JavaScript (d) PHP

Figure 12: Impact of generated code length on the extraction rate of generated code.

w/ TP - bear a striking resemblance to the output of w/o WM + w/o TP strategy. This case demonstrates that the watermarks we incorporated exert minimal influence on the output of LLM.

As depicted in Figure 13(c), w/ WM + w/o TP leads to meaningless generations due to the absence of grammar guidance, when w/ WM + w/ TP generates something similar to w/o WM + w/o TP, and also insert watermark "1012" into it. We posit that the observed outcome can be attributed to the fact that watermark logits have potentially increased the probability of erroneous type tokens being selected by LLM. Furthermore, it is discernible that once an incorrect type of token is chosen, the model's output will continually be misguided. For instance, upon the model's erroneous output of the token "`public`" due to the influence of watermark logits, it is anticipated that a complete function declaration will be subsequently generated by the model, thereby leading to a sustained impact on code semantics, and the generation of symbol "' ' '" will lead to the generation of comments, which also shows the misleading effect watermark logit have on code generation task. As shown in the Figure, when type predictor logit is applied, such circumstances are unlikely to occur.

Prompt

```
# Create an alias for a specific index/indices
def alias_create(indices, alias, hosts=None, body=None, profile=None, source=None):
```

w/o WM+w/o TP

```
if not hosts:
  hosts =
__salt__['elasticsearch.get_hosts'](profile)
  return
__salt__['elasticsearch.snapshot_get'](repository, snapshot, ignore_unavailable, hosts, profile)
  try:
    response =
es.snapshot.get(repository=repository, snapshot=snapshot,
ignore_unavailable=ignore_unavailable)
```

[...continued]

w. WM+w/o TP

```
# def snapshot_get(*, repository,
snapshot, ignore_unavailable=False)
  '''
  .. versionchanged:
  - In Elasticsearch 5.0 and later,
``ignoreUnavailable`` can be passed as a
URL parameter. If not provided, then it
defaults to ``True`` (which was
previously hard-coded in the client).
   .. versionchanged:
  - In Elasticsearch 6.0 and later,
``masterTimeout`` can be passed as a URL
parameter or as the value of
```

[...continued]

w. WM+w. TP

```
ret = {}

hosts =
__utils__["elasticsearch.check_cluster"]
(hosts)['master']

for host in hosts:
  try:
    url = "https://" + host +
"/_cat/snapshots/"+repository+" "+
snapshot
    headers =
{"accept":"application/json" }
```

[...continued]

(a) An example of Python code.

Prompt

```
// DeserializeIdentity provides a mock function with given fields: serializedIdentity
func (_m *IdentityDeserializer) DeserializeIdentity(serializedIdentity []byte)(msp.Identity, error) {
```

w/o WM+w/o TP

```
[Truncated...]
var r1 error
if rf, ok :=
retMock.Results[1].(func([]byte) error);
ok {
  r1 = rf(serializedIdentity)
  retMock.Results[1] = r1
} else {
    r1 = ret.Error(1)
  }
return r0, r1
}
```

w. WM+w/o TP

```
[Truncated...]
var r1 error
if rf, ok :=
ret_mock.call.Result.(func([]byte) error);
ok {
  r1 = rf(serializedIdentity)
  if r1 != nil || len(ret.Error(_i+0)) >
0 { // If there is an error, then it's
not nil
   return r0, errors.Wrap(err,
ret.Error(_i+0), 2, serialized)
}
```

w. WM+w. TP

```
[Truncated...]
var r1 error
if rf, ok := ret.Get(1).(func([]byte)
error); ok {
  if ret == reflect.Value{Zero}-
rf(args[0]) { //nolint: staticcheck
  }
  if ret == reflect.Value{Nil}-rf(args[0])
{ // nolint: staticcheck
  }
  r1 = ret.Error(1)
  return r1
}
```

(b) An example of Go code.

Prompt

```
// Creates the request protobuf. This method is considered an internal implementation detail and not meant to be used by applications.
public ReadRowsRequesttoProto(RequestContext requestContext) {
```

w/o WM+w/o TP

```
return Request.newBuilder()
  .setServiceName(serviceName)
  .setMethodName(methodName)
  .setPayload(ByteString.copyFrom(payloa
d))
  .build();
}
```

[continued...]

w. WM+w/o TP

```
public static final Request create(final
MethodDescriptor<?, ?> method, final
CallOptions callOptions);
```

```
### 3、`createResponse()`
```

```
public static final <RequestTypeT,
ResponseTypeT>
ServerCallListener.Listener<
```

[continued...]

w. WM+w. TP

```
Request.Builder builder =
Request.newBuilder();
builder.setMethod(getProtoMethod(requestT
ype));
builder.putAllHeaders(getProtoMap(getHead
ers()));
builder.putAllPathParamValues(getProtoMap
(getQueryParams()));
ByteString body = ByteString.EMPTY;
```

[continued...]

(c) An example of Java code.

Figure 13: Case study.