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]
Yulei Sui[4] **Pan Zhou**[1] **Lichao Sun**[5]
[1]Huazhong University of Science and Technology [2]University of Leeds
[3]Chongqing University [4]University of New South Wales [5]Lehigh University
{batuguan,wanyao,zqbi,panzhou}@hust.edu.cn z.wang5@leeds.ac.uk
hyzhang@cqu.edu.cn y.sui@unsw.edu.au lis221@lehigh.edu

Abstract

As Large Language Models (LLMs) are increasingly used to automate code generation, it is often desired to know if the code is AI-generated and by which model, especially for purposes like protecting intellectual property (IP) in industry and preventing academic misconduct in education. Incorporating watermarks into machine-generated content is one way to provide code provenance, but existing solutions are restricted to a single bit or lack flexibility. We present CodeIP, a new watermarking technique for LLM-based code generation. CodeIP enables the insertion of multi-bit information while preserving the semantics of the generated code, improving the strength and diversity of the inerseted watermark. This is achieved by training a type predictor to predict the subsequent grammar type of the next token to enhance the syntactical and semantic correctness of the generated code. Experiments on a real-world dataset across five programming languages showcase the effectiveness of CodeIP.

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 revolutionizing the way developers approaching programming by automatically generating code based on natural language intent and the context of surrounding code.

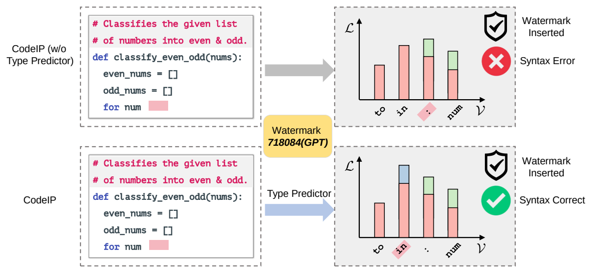
While LLMs have demonstrated great potential in automated code generation, they also raise challenges about safeguarding the intellectual property (IP) of the model architectures, weights, and training data due to the enormous cost of training a successful LLM (Li, 2024). Additionally, there are growing concerns in educational settings about academic integrity with the use of generative AI (Bozkurt et al., 2023). An important measure for protecting the LLM IP and preventing academic misconduct is the ability to determine if a piece of code is generated by a particular LLM.

Watermarking techniques (Kirchenbauer et al., 2023) offer a potential solution to determine the origin of machine-generated content. This technique is effective in safeguarding the IPs of Computer Vision (CV) and Natural Language Processing (NLP) models. It works by inserting information within multimedia formats (such as images and videos) without perceptibly diminishing the original utility of the content. By incorporating data such as owner/user ID, it supports leakage tracing, ownership identification, meta-data binding, and fortifying against tampering.

Existing watermarking techniques for language models can be categorized into two groups: hard and soft watermarks. A hard watermark is typically inserted by utilizing the masked language models (e.g., 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 performance. On the contrary, the 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).

Recently, several attempts have been made towards 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 gener-

[*]Corresponding Author.



```
# Classifies the given list
# of numbers into even & odd.
def classify_even_odd(nums):
  even_nums = []
  odd_nums = []
  for num
```

```
# Classifies the given list
# of numbers into even & odd.
def classify_even_odd(nums):
  even_nums = []
  odd_nums = []
  for num
```

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`".

ated code (Li et al., 2023b; Sun et al., 2023). However, a one-bit message can carry little information and is inadequate to preserve enough copyright information like the vendor ID of an LLM. Moreover, the implementation of a hard watermark does not offer robust protection, as the easily detectable nature of the hard-coded watermarking pattern undermines its effectiveness.

To this end, this paper puts forward a grammar-guided *multi-bit soft watermarking* method, termed CODEIP, to protect the IPs associated with LLMs during the code generation process. Specifically, following (Kirchenbauer et al., 2023), we first insert the watermark message based on the probability logit of LLMs during the code generation process. As this strategy has the potential to interfere with code semantics throughout the code generation process, we propose to incorporate grammar information into the process of generating watermarked 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 shows an example to illustrate the effectiveness of our introduced grammar information in comparison to the baseline model. In this example, our objective is to insert the multi-bit message (model name) "`718084`" (corresponding to the ASCII value of "`GPT`") into its generated code. It is evident that, in the absence of grammar guidance, the model 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 incorporating watermarks into a diverse real-world dataset that encompasses five programming languages, namely Java, Python, Go, JavaScript, and PHP. The experimental results validate the efficacy of our proposed approach to watermarking, demonstrating an average extraction rate of 0.95. Importantly, our approach maintains the utility of the generated code, exhibiting a 50% reduction in CodeBLEU losses compared to the baseline model that lacks grammar constraints.

This paper makes the following contributions.

- It is the first to study the problem of embedding the soft multi-bit watermarks into LLMs of code during the code generation process.
- It presents a new method that utilizes the grammatical information 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).

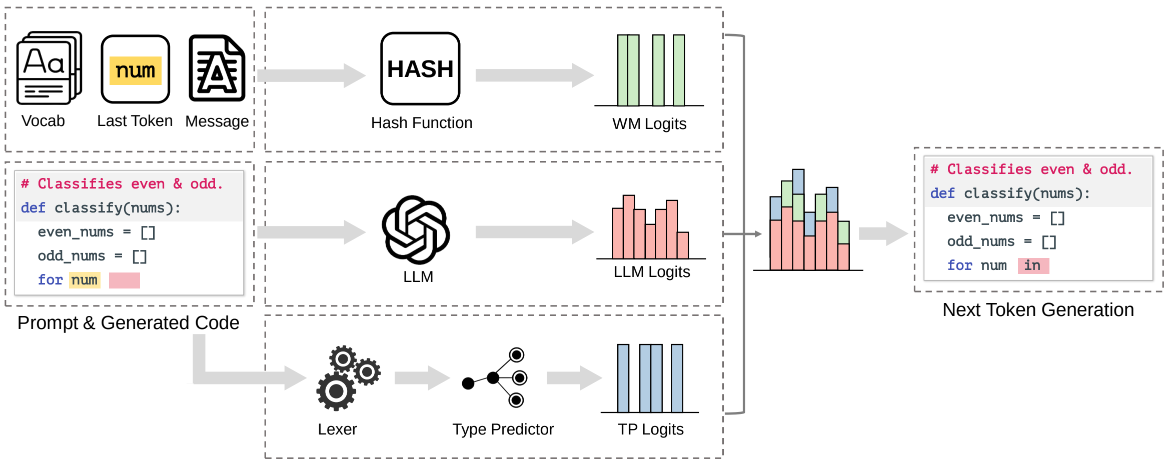
2 Preliminary

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 $\{w_1, w_2, \ldots, w_{|\rho|}\}$, where $|\cdot|$ denotes the length of a sequence. Let $\mathcal{V}$ denote the vocabulary used for mapping each token to corresponding indexes. Given a language model p_{LM}, the probability of the next token, conditioned on a prompt and a series of the previous generated tokens $w_{1:i}$, can be formulated as follows.

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

Here, $p_{\text{LM}}(w_i)$ denotes the probability distribution over the entire vocabulary $\mathcal{V}$, generated by the LM. We also call the probability distribution 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 source code, akin to the models in the GPT family, including Code Llama (Roziere et al., 2023) and



```
# Classifies even & odd.
def classify(nums):
    even_nums = []
    odd_nums = []
    for num
```

```
# Classifies even & odd.
def classify(nums):
    even_nums = []
    odd_nums = []
    for num  in
```

Figure 2: An overview of our proposed CodeIP.

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 multinomial sampling (Bengio et al., 2000) or greedy sampling (Berger et al., 1996). In this paper, we adopt the greedy sampling strategy. Therefore, the next token will be sampled based on the following equation: $w_i = \arg\max_{w \in \mathcal{V}} \log p_{\text{LM}}(w)$.

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 watermark inserting stage and the watermark 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 watermark stage, given an input snippet of code C, our expectation is 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

In Figure 2, we present an overview of our proposed CodeIP, which inserts a watermark into code generated by an LLM. The CodeIP comprises two distinct stages: watermark insertion, and grammar-guided watermarking. Initially, leveraging the decoding mechanism of existing LLMs, we use $\mathcal{L}_{\text{LM}}$ to denote the likelihood of each token in the vocabulary $\mathcal{V}$ as inferred by the LLM itself. Subsequently, during the watermark insertion stage (cf. Sec. 3.1), we incorporate the watermark message using a logit value $\mathcal{L}_{\text{WM}}$ calculated to measure its influence on $\mathcal{V}$. Moreover, we present a novel application of Context-Free Grammar (CFG) and introduce a logit (denoted as $\mathcal{L}_{\text{TP}}$), which signifies the probability associated with the grammatical type of the subsequent token, to guide the watermark insertion during the code generation process (cf. Sec. 3.2).

3.1 Watermark Insertion

Following Kirchenbauer et al. (2023), we insert the watermark into the generated code by modifying the probability distribution over the entire vocabulary $\mathcal{V}$ as the 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. Following Kirchenbauer et al. (2023), 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 probability of generation during the stage of 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 $[w_1, w_2, \cdots, w_{i-1}]$, 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 wi, preceded by the last token w_{i-1}, and denoting the watermark information as m, the watermark logit is computed as follows:

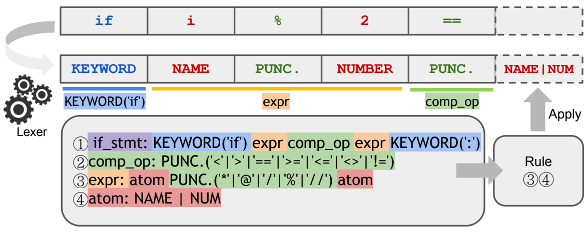
$$\begin{aligned}\mathcal{L}_{\text{WM}} &= \log p_{\text{WM}}\left(w_i \mid m, w_{i-1}\right) \\ &= \begin{cases} 1, & \mathcal{H}(w, m, w_{i-1}) = 1 \\ 0, & \mathcal{H}(w, m, w_{i-1}) = 0 \end{cases}\end{aligned} \tag{2}$$

Here h denotes a hash function which outputs a binarization value 0 or 1. p_{WM} denotes the probability distribution over the entire vocabulary $\mathcal{V}$, which conveys analogous implications to that of p_{LM}. 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

As previously mentioned, 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, for a segment of code, 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.



if
i
%
2
==
KEYWORD
NAME
PUNC.
NUMBER
PUNC.
NAME | NUM
Lexer
KEYWORD('if')
expr
comp_op
Apply
① if_stmt: KEYWORD('if') expr comp_op expr KEYWORD(':')
② comp_op: PUNC.('<'|'>'|'=='|'>='|'<='|'<>'|'!=')
③ expr: atom PUNC.('*'|'@'|'/'|'%'|'//') atom
④ atom: NAME | NUM
Rule ③④

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

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 as valid next token types, CFG's utility in aiding code generation tasks becomes significantly limited, for it lacks the capacity to calculate the probability distribution among these possible token types. Hence, we train a lexical token-type predictor and intend to utilize 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 forecasts a token type that will be outputted as the most probable lexical token type for the subsequent token.

In the context of code generation with LLMs and its prompt denoted as ρ, assuming that the LLM is in the process of generating the i-th token when the generated code denoted as G, for any given code snippet denoted as $S = [\rho; G_{1:i}]$, where $[\cdot;\cdot]$ denotes the concatenation of two elements, it is feasible to extract its token sequence $T = \text{Lexer}(S) = [\tau_1, \tau_2, \ldots, \tau_l]$ via lexical analysis, where $\tau \in \mathcal{T}$ denotes the lexical token type and l denotes the length of lexical token sequence. Then 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:

$$\tau_{l+1} = \mathrm{TP}(T) = \mathrm{LSTM}(\tau_1, \tau_2, \ldots, \tau_l)) \,. \tag{3}$$

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 $\mathcal{L}_{\mathrm{TP}}$.

The type predictor logit is a probability distribution of tokens within vocabulary $\mathcal{V}$. 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, $\mathcal{L}_{\mathrm{TP}}$ can be calculated as follows:

$$\mathcal{L}_{\mathrm{TP}} = \log p_{\mathrm{TP}}(w_i | [\rho; G_{1:i}]) = \begin{cases} 1, & w_i \in \Phi(\tau_{l+1}) \\ 0, & w_i \notin \Phi(\tau_{l+1}) \end{cases} \tag{4}$$

Here, ρ represents the prompt input into the LLM, and G denotes the pre-existing generated code. At this juncture, we are in the process of generating the i-th token w_i.

3.3 Combining the All

The subsequent section will present the watermark inserting formula corresponding to Figure 2, along with the ultimate watermark embedding algorithm.

$$w_i = \arg\max_{w \in \mathcal{V}} \{\mathcal{L}_{\mathrm{LM}} + \beta \mathcal{L}_{\mathrm{WM}} + \gamma \mathcal{L}_{\mathrm{TP}}\} \,. \tag{5}$$

In accordance with the settings established in the preceding sections, we posit that LLM is generating the i-th token. Herein, β and γ represent hyperparameters for LLM logit $\mathcal{L}_{\mathrm{LM}}$, watermark logit $\mathcal{L}_{\mathrm{WM}}$, and type predictor logit $\mathcal{L}_{\mathrm{TP}}$ respectively.

3.4 Watermark Extraction

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

$$m = \arg\max_{m'} \left\{ \sum_{i=1}^{L} \log p_{\mathrm{WM}} \left(w_i \mid m', w_{i-1} \right) \right\} \,, \tag{6}$$

where L denotes the length of the token sequence in the generated code G.

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., 2024) 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 computational 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 will 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 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

The default hyperparameters are configured as follows. 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. The type predictor is an LSTM model, which encompasses an embedding layer characterized 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,

Table 1: The result of watermark extraction rate. "WM": watermark, "TP": type predictor.

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

given the distinct grammatical structures inherent to each language. All the experiments in this paper 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 detected 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. 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. 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.

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 on most programming languages, indicating the efficacy of our watermarking techniques in the context of LLMs for code generation. Using DeepSeek Coder as a case in point, our watermarking strategy, both with and without the type predictor, demonstrates an impressive extraction rate of 0.99 for Java and 1.00 for PHP. These results are consistent 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 distinct logits ("w/ WM + w/o TP" and "w/ WM + w/ TP"), measured by the CodeBLEU score. From this table, it is evident that the exclusive use of watermark logit leads to a notable 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. Notably, in Java, Go, and JavaScript, the impact on CodeBLEU generation resulting from the simultaneous application of both logits (i.e., watermark logit and type predictor logit) is only half as pronounced as that arising solely from the use of watermark logit. This emphasizes the significant efficacy of the type predictor in preserving code semantics.

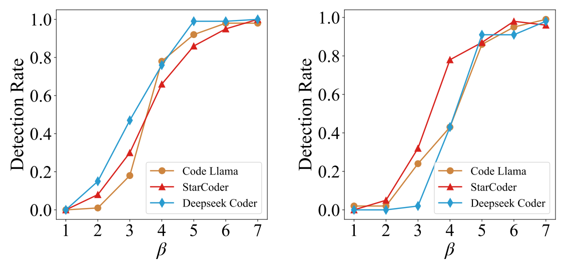
5.3 Parameter Analysis

The impact of parameter β. We conduct an experiment on the variation in extraction rates when adjusting the β value under three distinct LLMs for two programming languages, namely Java and Go, as shown 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, a 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 detected.

The impact of parameter γ. We conduct experiments on three distinct LLMs by varying the γ value, aiming to investigate the variations in extraction rates pertaining to two programming languages: Java and Go. The experimental results, as depicted in Figure 5, reveal a noteworthy trend. The initial augmentation of γ visibly improves the quality of the generated code. Nevertheless, as the augmentation progresses beyond a certain

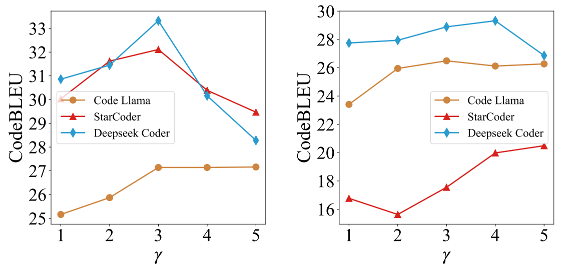
Table 2: CodeBLEU scores for different models with different strategies. The value in () represents the disparity in quality (CodeBLEU) between watermarked and non-watermarked code.

LLM	Strategy	Java	Python	Go	JavaScript	PHP
Code Llama	w/o WM + w/o TP	28.99	22.56	31.73	23.01	44.56
	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	w/o WM + w/o TP	39.16	17.74	27.61	24.06	42.60
	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	w/o WM + w/o TP	32.10	19.68	33.10	23.97	42.29
	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)



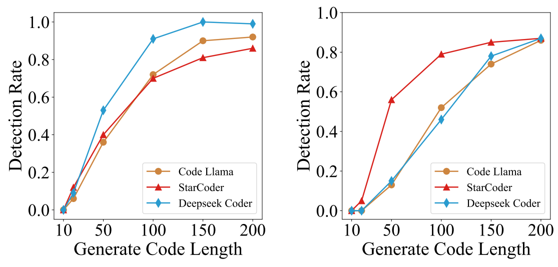
(a) Java (b) Go

Figure 4: Impact of parameter β.



(a) Java (b) Go

Figure 5: Impact of parameter γ.



(a) Java (b) Go

Figure 6: Impact of generated code length.

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. And 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 type of "`ran`" could potentially be classified as type "`NAME`", so the calculated lexical token type will be "`PUNCTUATION`", 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.

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 non-watermarked code, consequently presenting a heightened challenge in detecting watermarks within such 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 water-

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

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

mark—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 inherent robustness.

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; Manna and Waldinger, 1971). Recently, LLMs especially those pre-trained on code, such as DeepSeek Coder (Bi et al., 2024), 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), GitHub Copilot (Microsoft, 2024), and Tabnine (Tabnine, 2024), ushering in a new era of innovation.

Software Watermarking. The software watermarking problem has been studied since 1996 by Davidson and Myhrvold (1996), who altered code block 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) and Jiang et al. (2009). These rule-based early methods are often constrained by the usage scenarios and various attack techniques. Stern et al. (2000) has also proffered a methodology that entails the transformation and reorganization of code to uphold semantic integrity while concurrently resisting reverse engineering. Recently, several works (Yang et al., 2023; 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 present several limitations including their specificity for a single language, susceptibility to counterfeiting upon watermark method disclosure, restricting their applicability.

Machine Generated Text Identification. The task of identifying machine-generated text has always been of paramount importance. 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. In 2023, a novel method was introduced by Kirchenbauer et al. (2023) suggesting the inserting of watermarks into text during the model inference period. The authors applied a hash function and a random number generator to divide candidate tokens into groups, allowing watermark extraction by those aware of the rule. Lee et al. (2023) extended this method to code generation with 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 towards more secure LLM-powered software engineering through the continuation of our research.

8 Limitations

In our experiment, 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. 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.

References

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. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

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*.

Zetao Jiang, Rubing Zhong, and Bina Zheng. 2009. A software watermarking method based on public-key cryptography and graph coloring. In *2009 Third International Conference on Genetic and Evolutionary Computing*, pages 433–437. IEEE.

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*.

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*.

Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165.

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

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*.

Julien P Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. 2000. Robust object watermarking: Application to code. In *Information Hiding: Third International Workshop, IH'99, Dresden, Germany, September 29-October 1, 1999 Proceedings 3*, pages 368–378. Springer.

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.

Tabnine. 2024. Tabnine. https://www.tabnine.com/. [Online; accessed 1-Feb-2024].

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, 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.

Borui Yang, Wei Li, Liyao Xiang, and Bo Li. 2023. Towards code watermarking with dual-channel transformations. *arXiv preprint arXiv:2309.00860*.

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, a 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', 'TEXT', 'WHITESPACE',
'ESCAPE', 'ERROR', 'OTHER',
'KEYWORD', 'CONSTANT', 'DECLARATION',
'NAMESPACE', 'PSEUDO', 'RESERVED',
'TYPE', 'NAME', 'ATTRIBUTE',
'BUILTIN', 'PSEUDO', 'CLASS',
'CONSTANT', 'DECORATOR', 'ENTITY',
'EXCEPTION', 'FUNCTION', 'MAGIC',
'PROPERTY', 'LABEL', 'NAMESPACE',
'OTHER', 'TAG', 'VARIABLE', 'CLASS',
'GLOBAL', 'INSTANCE', 'MAGIC',
'LITERAL', 'DATE', 'STRING', 'AFFIX',
'BACKTICK', 'CHAR', 'DELIMITER',
'DOC', 'DOUBLE', 'ESCAPE', 'HEREDOC',
'INTERPOL', 'OTHER', 'REGEX',
'SINGLE', 'SYMBOL', 'NUMBER',
'BIN', 'FLOAT', 'HEX', 'INTEGER',
'LONG', 'OCT', 'OPERATOR', 'WORD',
'PUNCTUATION', 'MARKER', 'COMMENT',
'HASHBANG', 'MULTILINE', 'PREPROC',
'PREPROCFILE', 'SINGLE', 'SPECIAL',
'GENERIC', 'DELETED', 'EMPH',
'ERROR', 'HEADING', 'INSERTED',
'OUTPUT', 'PROMPT', 'STRONG',
'SUBHEADING', 'TRACEBACK'
```

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 articulated as follows.

$$\mathcal{J}(\mathcal{D}) = \sum_{d \in \mathcal{D}} \sum_{i=n}^{|T_d|} \log p_{\text{LSTM}}(l_i | l_{(i-n):i}) \tag{7}$$

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.

C CodeBLEU

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 $(\eta, \lambda, \mu, \xi) = (0.10, 0.10, 0.40, 0.40)$.

D Case Study

In Figure 7, 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 7(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 7(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 + 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 7(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 it is also 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'](profi
le)
  return
__salt__['elasticsearch.snapshot_get'](re
pository, 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.

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.

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.

Figure 7: Case study.