Marking Code Without Breaking It: Code Watermarking for Detecting LLM-Generated Code

Jungin Kim⋆ **Shinwoo Park**⋆ **Yo-Sub Han**†
Yonsei University, Seoul, Republic of Korea
jungin.kim@yonsei.ac.kr, pshkhh@yonsei.ac.kr, emmous@yonsei.ac.kr

Abstract

Identifying LLM-generated code through watermarking poses a challenge in preserving functional correctness. Previous methods rely on the assumption that watermarking high-entropy tokens effectively maintains output quality. Our analysis reveals a fundamental limitation of this assumption: syntax-critical tokens such as keywords often exhibit the highest entropy, making existing approaches vulnerable to logic corruption. We present STONE, a syntax-aware watermarking method that embeds watermarks only in non-syntactic tokens and preserves code integrity. For rigorous evaluation, we also introduce STEM, a comprehensive metric that balances three critical dimensions: correctness, detectability, and imperceptibility. Across Python, C++, and Java, STONE preserves correctness, sustains strong detectability, and achieves balanced performance with minimal computational overhead. Our implementation is available at https://github.com/inistory/STONE-watermarking.

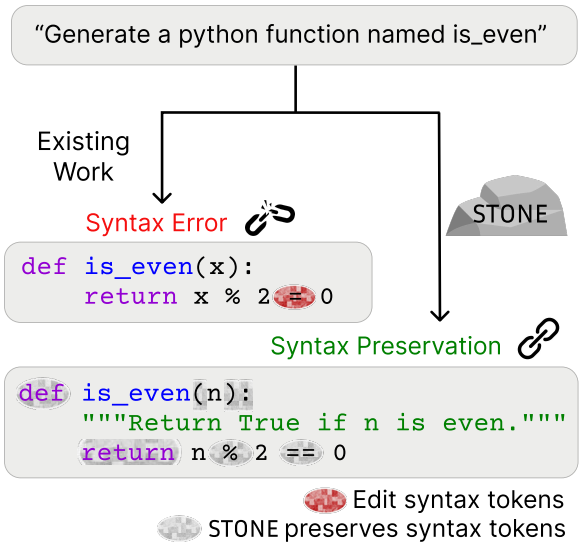

Figure 1: Motivating example of STONE watermarking. Existing methods modify syntax tokens and cause syntax errors, whereas STONE preserves syntax and embeds watermarks without breaking code structure.

1 Introduction

The rapid development of LLMs has significantly improved their ability to generate human-like code (Zan et al., 2022; Tang et al., 2023; Wei et al., 2023). These advancements have unlocked new possibilities, including software development and automated code generation (Nam et al., 2024; Wang et al., 2024; Guo et al., 2024a). However, this progress has raised new challenges in tracing the provenance of generated code (Yang et al., 2023; Li et al., 2023). As LLM-generated code becomes increasingly indistinguishable from human-written code, researchers have explored watermarking techniques to address this issue. As illustrated in Figure 1, existing approaches often modify syntax-critical tokens, which can introduce syntax errors or alter program behavior. In contrast, STONE preserves these tokens and embeds watermarks only into non-syntactic elements, thereby maintaining structural and functional integrity. LLM watermarking embeds specific patterns into the generated output during the generation process to help identify its origin (Sander et al., 2024; Golowich and Moitra, 2024; Hu and Huang, 2024) These patterns remain imperceptible to humans yet detectable through algorithmic analysis, thereby promoting greater transparency and accountability in AI-generated content (Guo et al., 2024b; Hou et al., 2024). Among various approaches, the green and red list-based watermarking is the most widely adopted (Kirchenbauer et al., 2023; Zhao et al., 2023; Lee et al., 2024; Lu et al., 2024). These approaches divide token candidates into two lists (green and red) and bias the selection process toward tokens in the green list. Detection involves

⋆ Authors equally contributed.
† Corresponding author.

Findings of the Association for Computational Linguistics: EACL 2026, pages 3990–4002
March 24-29, 2026

measuring the proportion of green list tokens in the generated output. While effective for natural language tasks, this approach does not easily apply to code generation, where structural and functional correctness are crucial (Lee et al., 2024; Hoang et al., 2024). Changes to syntax-critical tokens can cause compilation errors or alter program behavior.

We propose STONE (**S**yntax **TO**ke**N** preserving cod**E** watermarking), a method that selectively embeds watermarks in non-syntax tokens. By excluding tokens that are critical to code execution, STONE reduces the risk of functional degradation while embedding robust, detectable patterns. Additionally, we introduce STEM (Summative Test metric for Evaluating code-waterMarking). Existing studies on code watermarking rely on separate evaluation criteria, using different metrics and measurement protocols across methods, hindering fair and consistent comparison of performance. STEM integrates correctness, detectability, and imperceptibility into a unified metric, providing a fair basis for evaluating overall performance while allowing task-specific weighting. We employ STEM to evaluate STONE and compare it with state-of-the-art watermarking methods. Experiments across Python, C++, and Java show that STONE preserves functional correctness, achieves balanced performance across all evaluation criteria, and consistently outperforms prior methods in efficiency.

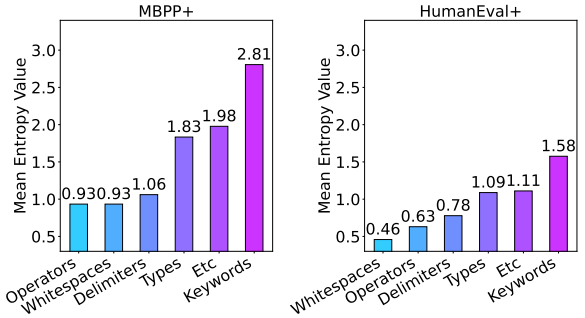

Figure 2: Entropy values by token type. Tokens that do not fall under keywords, whitespace, types, delimiters, or operators are categorized as *etc* tokens. The *etc* category is the primary target for our proposed STONE watermarking method. Token entropy is measured using Qwen2.5-Coder-7B.

2 Motivation and Preliminary Analysis

The recent code watermarking method SWEET (Lee et al., 2024) embeds watermarks in high-entropy tokens. This design is motivated by the intuition that modifying tokens with higher uncertainty is less likely to affect the perceived quality of the generated code. In our preliminary analysis, we examine whether token entropy indeed varies systematically across syntactic categories and how such variation relates to potential impacts on code quality.

We consider two popular benchmarks, MBPP+ and HumanEval+, for Python code generation and completion. Each Python token is classified into one of five syntactic categories (keywords, whitespace, types, delimiters, and operators), with all remaining tokens grouped into an *etc* category. Details of the categorization procedure are provided in Appendix C.

We compute token entropy for each category using Shannon entropy, following Lee et al. (2024). Given a token sequence $y = (y_0, y_1, \ldots, y_N)$, the entropy at generation step t is defined as:

$$H_t = -\sum_{i=1}^{|V|} P(y_t = v_i \mid y_{<t}) \log P(y_t = v_i \mid y_{<t}), \tag{1}$$

where V denotes the vocabulary and $y_{<t}$ the previously generated tokens. This formulation captures the uncertainty of the model's next-token prediction at each generation step.

Figure 2 shows the mean entropy values for each token category. We observe that keywords exhibit the highest average entropy among all categories. This can be attributed to their diverse roles across control flow, structural definitions, and other syntactic constructs in Python. In contrast, operators, delimiters, types, and whitespace follow more regular and constrained syntactic patterns, resulting in comparatively lower entropy.

Although keywords have high entropy, watermarking them risks altering tokens that are essential to code syntax and semantics. Even small perturbations to such tokens may lead to functional errors or noticeable degradation in code quality. In comparison, the *etc* category contains tokens that are less critical to core syntactic structure while still exhibiting relatively high entropy. This observation suggests that embedding watermarks in non-syntactic tokens may preserve code quality better than approaches that rely solely on high-entropy tokens.

3 Methodology

3.1 Syntax-Aware Watermarking: STONE

STONE is a code watermarking method designed to preserve the quality of generated code. This section introduces the watermark insertion and detection procedures of STONE.

Inserting Watermarks We treat keywords, whitespace, types, delimiters, and operators as syntactic elements and embed watermarks only in non-syntactic tokens. Algorithm 1 describes how STONE embeds watermarks during code generation. Given a prompt $\mathbf{x}$ and previously generated tokens $\mathbf{y}_{<t}$, the language model f_{LM} computes the initial probability distribution $\mathbf{p}_t$ at time step t. STONE first samples a candidate token $\tilde{y}_t$ from the initial probability distribution. If the sampled token is not in the syntax element set, STONE then divides the vocabulary $\mathcal{V}$ into a green and red list according to the green token ratio γ. It then increases the likelihood of sampling green-list tokens by adding a constant δ to their logit values. Using the adjusted logits, STONE updates the probability distribution and samples the final token y_t for time step t.

Algorithm 1 Insertion Algorithm of STONE

Require: Prompt $\mathbf{x}$, start token y_0, last generated tokens $\mathbf{y}_{<t} = (y_0, \ldots, y_{t-1})$, vocabulary set $\mathcal{V}$, language model f_{LM}, syntax element set S, green list ratio $\gamma \in (0, 1)$, logit adjustment factor $\delta > 0$

for $t = 1, 2, 3, \ldots$ **do**
  Compute the logit vector:

$$\mathbf{l}_t = f_{\text{LM}}(\mathbf{x}, \mathbf{y}_{<t})$$

  Compute the initial probability vector $\mathbf{p}_t$:

$$p_{t,i} = \frac{e^{\mathbf{l}_t[i]}}{\sum_{j=1}^{|\mathcal{V}|} e^{\mathbf{l}_t[j]}}, \quad \text{for } i \in \{1, \ldots, |\mathcal{V}|\}$$

  Sample a candidate token $\tilde{y}_t \sim \mathbf{p}_t$
  if $\tilde{y}_t \notin S$ **then**
    Compute a hash of token y_{t-1} as a seed
    Split vocabulary indices into $\mathcal{G}_t$ and $\mathcal{R}_t$
    Increase logits for indices in $\mathcal{G}_t$ by δ:

$$\mathbf{l}_t[i] \leftarrow \mathbf{l}_t[i] + \delta, \quad \text{for } i \in \mathcal{G}_t$$

    Recompute the adjusted probability vector $\mathbf{p}'_t$:

$$p'_{t,i} = \frac{e^{\mathbf{l}_t[i]}}{\sum_{j=1}^{|\mathcal{V}|} e^{\mathbf{l}_t[j]}}, \quad \text{for } i \in \{1, \ldots, |\mathcal{V}|\}$$

  else
    Set $\mathbf{p}'_t = \mathbf{p}_t$
  end if
  Sample the final token $y_t \sim \mathbf{p}'_t$
end for

Detecting Watermarks We assess how many green tokens are present in the code, and if this value surpasses a certain threshold, we conclude that the code was generated by an LLM. When identifying potential green tokens, we focus only on non-syntactic elements. Algorithm 2 details the watermark detection process. N^E represents the number of *etc* tokens (non-syntax tokens) in the code, and N_G^E denotes the number of green tokens among them. We replicate the division of the vocabulary into green and red lists at each time step during code generation to determine whether a specific token is a green token. If the z-score exceeds the predefined threshold $z_{\text{threshold}}$, the code is determined to be generated by an LLM.

Algorithm 2 Detection Algorithm of STONE

Require: Token sequence $\mathbf{X} = (X_0, \ldots, X_{N-1})$, syntax element set S, green list ratio $\gamma \in (0, 1)$, z-score threshold $z_{\text{threshold}} > 0$

Initialize $N^E \leftarrow 0$, $N_G^E \leftarrow 0$
for $t = 1, 2, \ldots, N-1$ **do**
  if $X_t \notin S$ **then**
    $N^E \leftarrow N^E + 1$
    Compute a hash of token X_{t-1} as a seed
    Split vocabulary indices into $\mathcal{G}_t$ and $\mathcal{R}_t$
    if $X_t \in \mathcal{G}_t$ **then**
      $N_G^E \leftarrow N_G^E + 1$
    end if
  end if
end for
Compute the z-score:

$$z = \frac{N_G^E - \gamma N^E}{\sqrt{\gamma(1-\gamma)N^E}}$$

if $z > z_{\text{threshold}}$ **then**
  return True ▷ Sequence $\mathbf{X}$ is watermarked
else
  return False ▷ Sequence $\mathbf{X}$ is not watermarked
end if

3.2 Unified Evaluation Metric: STEM

Prior studies on code watermarking typically assess functional correctness, detectability, and imperceptibility independently (Lee et al., 2024; Yang et al., 2024; Li et al., 2024; Guan et al., 2024). They adopt diverse definitions and evaluation protocols, which prevents consistent and fair comparison across methods. For instance, a technique may improve detectability at the cost of correctness, leading to inconsistent evaluations. STEM addresses this issue by unifying the three criteria into a single weighted metric (α, β, ζ), where $\alpha + \beta + \zeta = 1$, enabling fair and direct comparison across methods under configurable trade-offs.

$$\begin{aligned}\texttt{STEM} = \alpha \cdot \text{Correctness}(C_{wm}) \\ + \beta \cdot \text{Detectability}(C_{wm}, C_H) \\ + \zeta \cdot \text{Imperceptibility}(C_{wm}, C).\end{aligned} \tag{2}$$

Correctness *Correctness* is crucial for preventing watermarking from compromising code functionality. Let C_{wm} denote the set of watermarked code samples. We measure functionality preservation using the unbiased estimator of pass@k (Chen et al., 2021), which is standard for evaluating code generation correctness:

$$\text{Correctness}(C_{wm}) = \mathbb{E}_{C_{wm}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right], \tag{3}$$

where n corresponds to the total number of generated solutions for a task, and c denotes the subset of those solutions that pass all test cases. This estimator measures the probability that at least one of the k generated solutions is functionally correct. In the context of watermarking, it captures whether watermark insertion introduces execution failures or alters program semantics. A higher correctness score indicates that watermark insertion does not hinder program execution.

Detectability *Detectability* measures how effectively a watermark can be identified in generated code. We assess detectability using a z-score that quantifies the proportion of green-list tokens, and aggregate scores by computing the AUROC between human-written and watermarked corpora.

For a generated sequence $X_{wm} = (X_0, X_1, \ldots, X_{N-1})$, the z-score of green-token ratio is defined as

$$z(X_{wm}) = \frac{|X_{wm}|_G - \gamma|X_{wm}|}{\sqrt{\gamma(1-\gamma)|X_{wm}|}}, \tag{4}$$

where $|X_{wm}|_G$ is the number of green tokens in X_{wm} and γ is the expected green-token ratio in non-watermarked code. Higher z values indicate stronger evidence that the code is watermarked. The z-score serves as a continuous detection statistic, enabling threshold-based discrimination between watermarked and non-watermarked sequences. By varying the threshold τ, different operating points in terms of true and false positive rates are obtained, forming the ROC curve.

Given a corpus of human-written code $C_H = \{X_H^{(i)}\}$ and watermarked code $C_{wm} = \{X_{wm}^{(j)}\}$, the true positive rate (TPR) and false positive rate (FPR) at threshold τ are defined as follows:

$$\text{TPR}(\tau) = \frac{\sum_{j=1}^{J} \mathbf{1}\big[z(X_{wm}^{(j)}) > \tau\big]}{J}. \tag{5}$$

$$\text{FPR}(\tau) = \frac{\sum_{i=1}^{I} \mathbf{1}\big[z(X_H^{(i)}) > \tau\big]}{I}. \tag{6}$$

The area under the ROC curve (AUROC) is then

$$\text{Detectability}(C_{wm}, C_H) = \int_0^1 \text{TPR}(\text{FPR})\, d(\text{FPR}), \tag{7}$$

where higher AUROC values indicate stronger separability between watermarked and human-written code.

Imperceptibility *Imperceptibility* concerns the preservation of token probability distributions consistent with normal model outputs (Huang and Wan, 2025). Since code LLMs follow stable and highly structured token distributions, distortions introduced by watermarking can form statistical signals that are exploitable by adversaries without access to the secret key. Accordingly, following perplexity-based approaches in text watermarking (Kirchenbauer et al., 2023) and studies on code generation (Magnusson et al., 2024), we define imperceptibility as the deviation in token probability distributions introduced by watermarking, measured through perplexity shifts computed with code LLMs.

Let C_{wm} and C denote the corpora of watermarked and non-watermarked code, respectively. Perplexity is computed over the watermarked corpus $C_{wm} = \{X_{wm}^{(1)}, \ldots, X_{wm}^{(J)}\}$ as:

$$\text{PPL}(C_{wm}) = \frac{1}{|C_{wm}|} \sum_{j=1}^{|C_{wm}|} \exp\Big(-\mathbb{E}_i\Big[\log P\Big(y_i^{(j)} \mid y_{<i}^{(j)}\Big)\Big]\Big), \tag{8}$$

where $P(y_i^{(j)}|y_{<i}^{(j)})$ represents the probability of token $y_i^{(j)}$ given its preceding context. An effectively concealed watermark minimizes perplexity shifts, ensuring imperceptibility. We define the imperceptibility metric as:

$$\text{Imperceptibility}(C_{wm}, C) = 1 - \frac{\big|\text{PPL}(C_{wm}) - \text{PPL}(C)\big|}{\text{PPL}(C)}. \tag{9}$$

This metric quantifies the perplexity-based difference between LLM-generated code with watermark (C_{wm}) and non-watermarked code (C). It computes the relative difference by normalizing the absolute gap by the perplexity of C, providing a scale-invariant measure of distributional shift. Higher values indicate minimal impact on token distributions and thus stronger imperceptibility, while lower or negative values reflect greater disruption due to watermark insertion.

4 Experimental Setup

Datasets. For our experiments, we evaluate our approach across three programming languages: Python, C++, and Java. We use four benchmark datasets: HumanEval+ and MBPP+ (Liu et al., 2023) for Python, and HumanEvalPack (Muennighoff et al., 2024) for C++ and Java. Table 1 reports token-length statistics of the evaluation datasets, based on the Qwen2.5-Coder-7B tokenizer. The datasets span from short snippets to long sequences, allowing evaluation under varying code lengths and structural complexity.

Dataset	Problems	Test Cases	Max	Min	Mean	Std.
MBPP+	399	105	341	11	40.25	34.91
HumanEval+	164	764	558	43	188.28	87.32
HEP-C++	164	764	697	42	223.10	107.57
HEP-Java	164	764	660	56	237.36	105.09

Table 1: Token length statistics of solution code across evaluation datasets, along with the total number of problems and the average number of test cases per problem.

Baselines. We consider the following three watermarking approaches for comparison. (1) KGW (Kirchenbauer et al., 2023) generates text by dividing the vocabulary into a green and red list at each time step of token generation, increasing the probability of generating tokens from the green list. (2) EWD (Lu et al., 2024) leverages entropy-based token weighting during detection, giving more influence to high-entropy tokens, thus enhancing detection in texts with varying entropy distributions. (3) SWEET (Lee et al., 2024) selectively embeds watermarks in high-entropy tokens, addressing detection challenges posed by the low entropy in code. We exclude CodeIP (Guan et al., 2024) from our baselines as it requires additional training for watermark insertion, while our approach and the selected baselines are training-free methods. As part of our comparative analysis, we reproduce CodeIP and report its experimental results in Appendix A.

Base Model. We use Qwen2.5-Coder-7B (Hui et al., 2024) for our experiments. We also report additional results with Llama-3.1-8B in Appendix B, which further demonstrate the generalizability of our approach across different model architectures.

Evaluation Metric. We evaluate watermarking methods across three dimensions: Correctness, Detectability, and Imperceptibility. Correctness uses pass@k (k=1,5), detectability is measured by AUROC with a z-score test, and imperceptibility by perplexity computed with StarCoder2-7B. We use the STEM metric to integrate these three axes into a single composite score. For the main results (Tables 2 and 3), we adopt an equal-weight configuration $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. We further analyze weight settings with alternative configurations in Appendix D.

Implementation Details. During code generation, we apply top-k sampling with $k = 50$, restricting the candidate token pool to the top 50 tokens with the highest logits and normalizing their probabilities before sampling. The temperature, which controls the sharpness of the token distribution, is fixed at 1.0. We set $\gamma = 0.5$ and $\delta = 1.0$ for the MBPP+ dataset, while for the HumanEval+ and HumanEvalPack datasets (C++ and Java), we set $\gamma = 0.5$ and $\delta = 0.5$. We conduct all experiments on an NVIDIA A6000 GPU.

5 Experiments and Results

We pose three research questions and provide a thorough analysis of each:

- **RQ1:** Can STONE preserve functional correctness?
- **RQ2:** Does STONE achieve balanced performance across correctness, detectability, and imperceptibility as measured by STEM?
- **RQ3:** How efficient is STONE in watermark insertion and detection?

We address these questions through three analytical dimensions: (1) **Functionality Preservation**, which verifies that watermark insertion does not alter program behavior; (2) **Balanced Performance under STEM**, which examines whether STONE maintains a well-balanced trade-off among correctness,

Method	MBPP+				HumanEval+			
	Correctness	Detectability	Imperceptibility	STEM	Correctness	Detectability	Imperceptibility	STEM
KGW	0.499	0.831	**0.994**	0.775	0.573	0.523	**0.986**	0.694
EWD	0.499	0.965	**0.994**	0.819	0.573	0.730	**0.986**	0.763
SWEET	0.502	0.867	0.992	0.787	0.574	0.710	0.978	0.754
STONE	**0.571**	**0.982**	0.990	**0.848**	**0.587**	**0.777**	0.978	**0.781**

Table 2: Experimental results on MBPP+ and HumanEval+. Metrics include correctness, detectability, imperceptibility, and the integrated STEM score under the equal-weight configuration.

Method	HEP-C++				HEP-Java			
	Correctness	Detectability	Imperceptibility	STEM	Correctness	Detectability	Imperceptibility	STEM
KGW	0.576	0.621	**0.993**	0.730	0.387	0.546	**0.993**	0.642
EWD	0.576	0.681	**0.993**	0.750	0.387	0.646	**0.993**	0.675
SWEET	0.584	0.641	0.979	0.735	0.413	0.580	0.901	0.631
STONE	**0.622**	**0.729**	0.990	**0.780**	**0.445**	**0.721**	0.979	**0.715**

Table 3: Experimental results on HumanEvalPack-C++ and HumanEvalPack-Java. Metrics include correctness, detectability, imperceptibility, and the integrated STEM score under the equal-weight configuration.

detectability, and imperceptibility; and (3) **Efficiency**, which measures the computational overhead of watermark insertion and detection.

RQ1 We examine whether STONE embeds watermarks without affecting the functional correctness of generated code. Tables 2 and 3 show that STONE improves correctness by 7.57% over SWEET on average across all benchmarks. The token-level analysis in Appendix E indicates that the entropy-based token selection method used by SWEET often chooses syntax-related elements—such as delimiters, whitespace, keywords, types, and operators—that are essential for correct execution. For example, modifying delimiters like colons (':') or brackets ('', '[]') can cause parsing errors, and changing operators or keywords (e.g., replacing '+' with '-' or 'True' with 'False') can alter program logic or control flow. Even under the optimal configuration, syntax tokens account for 12.6% of the tokens selected for watermarking in the HumanEval+ dataset, with delimiters and whitespace forming the majority. This inclusion of syntax elements explains the observed reduction in correctness for SWEET. In contrast, STONE excludes syntax tokens and embeds watermarks only into non-syntactic elements, which prevents structural and logical errors during execution and allows the generated code to preserve its functional behavior.

RQ2 We examine whether STONE achieves balanced performance across correctness, detectability, and imperceptibility. Baselines such as KGW and EWD watermark every token, which dilutes the watermark signal and lowers AUROC scores. SWEET modifies high-entropy tokens, including syntax elements, improving detectability but reducing syntactic stability. In contrast, STONE embeds watermarks only in non-syntax tokens, preserving syntactic integrity while keeping the signal distinct. This selective design enables STONE to achieve higher detection accuracy without compromising code imperceptibility.

Imperceptibility results reinforce this finding. KGW and EWD maintain reasonable imperceptibility by uniformly adjusting token probabilities, but their adjustments can still introduce subtle artifacts. SWEET often produces more noticeable distributional shifts because it alters syntax tokens. STONE, in contrast, maintains low perplexity differences by leaving syntax untouched and modifying only non-syntax tokens. As a result, STONE maintains imperceptibility without sacrificing correctness or detection robustness. We quantify this balanced behavior using the STEM metric, which under the equal-weight configuration provides a neutral evaluation across correctness, detectability, and imperceptibility. Unlike other watermarking methods that exhibit clear trade-offs among these criteria, STONE achieves consistently strong overall performance. A comprehensive grid analysis across 66 weight settings (Appendix D) further confirms that this advantage remains stable and does not depend on any specific weighting scheme.

RQ3 We analyze the computational overhead of watermarking in terms of insertion and detec-

Method	MBPP+		HumanEval+		HEP-C++		HEP-Java	
	Insertion	Detection	Insertion	Detection	Insertion	Detection	Insertion	Detection
KGW	3320	12.90	1268	4.19	1308	8.01	1506	8.95
EWD	3320	100.43	1268	34.51	1308	56.85	1506	59.84
SWEET	3300	100.94	1270	34.68	1454	58.08	1480	59.22
STONE	3266	13.27	1277	4.62	1300	8.48	1459	9.24

Table 4: Comparison of total insertion and detection time (measured in seconds) across different watermarking methods. The reported measurement is performed at the dataset level.

tion time. Insertion time denotes the duration required to generate watermarked code, whereas detection time measures the time needed to verify a watermark. Table 4 summarizes the results for each method. All approaches exhibit comparable insertion times, but STONE achieves detection speeds that are on average 86% faster than those of SWEET and EWD at the dataset level. This improvement results from differences in detection mechanisms. SWEET and EWD employ entropy-based approaches that compute token probability distributions, which substantially increase computational overhead. In contrast, STONE verifies watermarks using a predefined green list of non-syntax tokens, reducing detection time while maintaining reliable detectability.

6 Analysis

6.1 Trade-off Between Code Quality and Detection Performance

We analyze the trade-off between code quality and watermark detectability as influenced by two watermarking parameters. The parameter γ controls the proportion of the vocabulary designated as the green list, while δ determines the extent to which the selection probability of green tokens is amplified. We conduct an analysis on the HumanEval+ dataset by varying the values of δ and γ. Figure 3 shows the results in a bubble chart, revealing a clear trade-off between code quality and detectability. The circular markers represent the pass@1 and detection performance (AUROC) of STONE, while the triangular markers indicate those of SWEET. Among the four regions divided by the two black dashed lines, the upper-right quadrant corresponds to configurations where both code quality and detection performance are high, indicating a desirable balance between utility and detectability. We observe that, while only a single configuration of SWEET falls into this balanced region, multiple configurations of STONE—spanning various combinations of γ and δ—lie within it. This indicates that STONE achieves a more balanced trade-off between functionality and watermark detectability compared to SWEET.

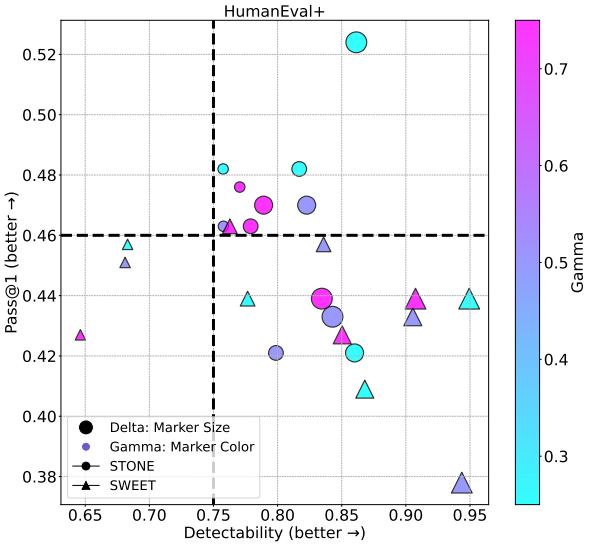

Figure 3: The trade-off between pass@1 (Y-axis) and detectability (X-axis). We analyze the impact of the green list ratio γ (indicated by marker color) and the watermark strength δ (indicated by marker size). Circular and triangular markers represent STONE and SWEET, respectively.

6.2 Robustness under Adversarial Attacks

Although a watermark is embedded in the code, a malicious user may remove it by modifying the code. Therefore, the watermark must remain robust against attacks like code modifications. We assess the resilience of watermarks of STONE against code alterations by applying two types of attacks: (1) a code refactoring attack using a commercial service[1] and (2) a code paraphrasing attack using ChatGPT (GPT-4o)[2]. We conduct this analysis on the HumanEval+ dataset. Figure 4 shows the classification performance (i.e., detectability) in dis-

[1] https://codepal.ai/code-refactor

[2] Prompt: Please paraphrase the following code.

tinguishing between watermarked LLM-generated code and human-written code after these attacks are applied to code watermarked by STONE and SWEET. The experimental results show that STONE is more robust than SWEET against both types of attacks. Refactoring reorganizes code while keeping syntax boundaries, so STONE preserves many watermarks inserted into non-syntax tokens. Paraphrasing changes variable names and expressions, which directly correspond to the embedding targets of STONE, and thus decreases detectability. SWEET shows the opposite tendency because refactoring alters its watermarking targets. STONE maintains higher robustness under both attacks by embedding watermarks into a syntax-filtered token space that remains stable under structural and lexical modifications. This result confirms that STONE maintains consistent detection robustness across different attack types.

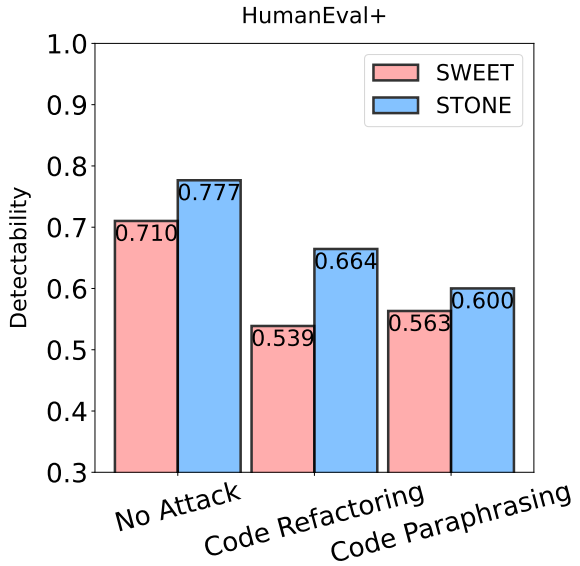

Figure 4: We evaluate robustness by applying code refactoring and paraphrasing attacks to watermarked code and measuring detection performance. Results compare the resilience of SWEET and STONE under both attacks.

7 Related Work

Post-hoc detection methods (Xu and Sheng, 2024; Shi et al., 2024) identify machine-generated code after generation using statistical cues such as perplexity or syntax differences. In contrast, watermarking embeds provenance signals during generation to enable direct identification (Kirchenbauer et al., 2023; Guan et al., 2024; Park et al., 2025). Our work focuses on the latter approach, developing a syntax-aware watermarking method for code generation.

Among watermarking approaches, Lee et al. (2024) proposed SWEET, a method that selectively embeds watermarks in high-entropy tokens. While SWEET demonstrated improvements in detection capability and code quality, our preliminary analysis revealed that approximately 12.6% of these high-entropy tokens correspond to syntax elements such as reserved keywords. Modifying these critical tokens can disrupt program logic, resulting in degraded functional correctness. Guan et al. (2024) introduced CodeIP, a grammar-guided watermarking approach by utilizing type prediction to maintain syntactic validity. This method ensured that the generated code remained grammatically correct. However, the reliance on type prediction requires significant computational resources, limiting its practicality for large-scale code generation tasks. In contrast to these approaches, our method explicitly excludes syntax-critical tokens from watermark insertion. Our approach ensures that essential structural elements, like keywords and operators, remain unchanged, thereby preserving functional correctness without additional computational overhead. ACW (Li et al., 2024) and SrcMarker (Yang et al., 2024) employed watermarking techniques based on code transformations, such as semantic-preserving edits and variable renaming. While these approaches preserved program functionality, the resulting modifications often introduce unnatural code patterns, which can increase the risk of watermark removal by adversaries.

8 Conclusion

We present STONE, a syntax-aware watermarking method that preserves functional correctness by excluding syntax-critical tokens during watermark insertion. We also propose STEM, a unified evaluation metric integrating correctness, detectability, and imperceptibility for balanced assessment. Experiments on Python, C++, and Java show that STONE outperforms baseline methods without degrading code quality and reduces watermark detection time compared with entropy-based approaches. Evaluation under the STEM metric demonstrates balanced performance across weight configurations, and further analysis confirms that STONE maintains stable and robust performance under code modifications.

Limitations

While STONE provides a robust approach to functionality-preserving watermarking, it has several limitations.

First, the watermarking capacity is based on the density of non-syntactic tokens. The conventional coding styles in our benchmark datasets provided ample opportunities for watermark insertion. This efficacy, however, can be compromised in unconventional code, such as obfuscated scripts or solutions for code golf, where such tokens are sparse. Under these conditions, the embedded watermark may prove too sparse for reliable detection.

Second, although our evaluation demonstrates resilience to general attacks like paraphrasing and refactoring, a more sophisticated threat model considers an adversary with specific knowledge of the STONE algorithm. Such an adversary may directly attack non-syntactic tokens by systematically renaming all variables and removing comments. This class of attack poses a more significant threat to watermark integrity than general modifications. Developing countermeasures against such targeted, algorithm-aware attacks remains an important direction for future research.

Acknowledgments

This research was supported by the NRF grant (RS-2025-00562134) and the AI Graduate School Program (RS-2020-II201361) funded by the Korean government.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Noah Golowich and Ankur Moitra. 2024. Edit distance robust watermarks via indexing pseudorandom codes. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Batu Guan, Yao Wan, Zhangqian Bi, Zheng Wang, Hongyu Zhang, Yulei Sui, Pan Zhou, and Lichao Sun. 2024. Codeip: A grammar-guided multi-bit watermark for large language models of code. In *Findings of the Association for Computational Linguistics: EMNLP 2024*.

Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu. 2024a. Ft2ra: A fine-tuning-inspired approach to retrieval-augmented code completion. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*. ACM.

Yuxuan Guo, Zhiliang Tian, Yiping Song, Tianlun Liu, Liang Ding, and Dongsheng Li. 2024b. Context-aware watermark with semantic balanced green-red lists for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*. Association for Computational Linguistics.

Duy C Hoang, Hung TQ Le, Rui Chu, Ping Li, Weijie Zhao, Yingjie Lao, and Khoa D Doan. 2024. Less is more: Sparse watermarking in llms with enhanced text quality. *arXiv preprint arXiv:2407.13803*.

Abe Bohan Hou, Jingyu Zhang, Tianxing He, Yichen Wang, Yung-Sung Chuang, Hongwei Wang, Lingfeng Shen, Benjamin Van Durme, Daniel Khashabi, and Yulia Tsvetkov. 2024. Semstamp: A semantic watermark with paraphrastic robustness for text generation. 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*. Association for Computational Linguistics.

Zhengmian Hu and Heng Huang. 2024. Inevitable trade-off between watermark strength and speculative sampling efficiency for language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Baizhou Huang and Xiaojun Wan. 2025. WaterPool: A language model watermark mitigating trade-offs among imperceptibility, efficacy and robustness. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2023. A Watermark for Large Language Models. In *International Conference on Machine Learning*.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. 2024. Who wrote this code? watermarking for code generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Boquan Li, Mengdi Zhang, Peixin Zhang, Jun Sun, and Xingmei Wang. 2024. Resilient watermarking for llm-generated codes. *arXiv preprint arXiv:2402.07518*.

Zongjie Li, Chaozheng Wang, Shuai Wang, and Cuiyun Gao. 2023. 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*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yijian Lu, Aiwei Liu, Dianzhi Yu, Jingjing Li, and Irwin King. 2024. An entropy-based text watermarking detection method. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Ian Magnusson, Akshita Bhagia, Valentin Hofmann, Luca Soldaini, Ananya Harsh Jha, Oyvind Tafjord, Dustin Schwenk, Evan Walsh, Yanai Elazar, Kyle Lo, and 1 others. 2024. Paloma: A benchmark for evaluating language model fit. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *International Conference on Learning Representations (ICLR), ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Shinwoo Park, Hyejin Park, Hyeseon Ahn, and Yo-Sub Han. 2025. Watermod: Modular token-rank partitioning for probability-balanced llm watermarking. *arXiv preprint arXiv:2511.07863*.

Tom Sander, Pierre Fernandez, Alain Durmus, Matthijs Douze, and Teddy Furon. 2024. Watermarking makes language models radioactive. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. 2024. Between lines of code: Unraveling the distinct patterns of machine and human programmers. *CoRR*, abs/2401.06461.

Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain adaptive code completion via language models and decoupled domain databases. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE.

Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching code llms to use autocompletion tools in repository-level code generation. *arXiv preprint arXiv:2401.06391*.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM.

Zhenyu Xu and Victor S. Sheng. 2024. Detecting ai-generated code assignments using perplexity of large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

B. Yang, W. Li, L. Xiang, and B. Li. 2024. Srcmarker: Dual-channel source code watermarking via scalable code transformations. In *2024 IEEE Symposium on Security and Privacy (SP)*.

Xianjun Yang, Liangming Pan, Xuandong Zhao, Haifeng Chen, Linda R. Petzold, William Yang Wang, and Wei Cheng. 2023. A survey on detection of llms-generated content. *CoRR*, abs/2310.15654.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large Language Models Meet NL2Code: A Survey. *arXiv preprint arXiv:2212.09420*.

Xuandong Zhao, Prabhanjan Ananth, Lei Li, and Yu-Xiang Wang. 2023. Provable robust watermarking for ai-generated text. *arXiv preprint arXiv:2306.17439*.

A Performance Analysis of CodeIP

Dataset	Correct.	Detect.	Imperc.	PPL (NW→WM)
MBPP+	0.093	0.962	-0.246	(3.504→7.869)
HumanEval+	0.018	0.945	-0.075	(3.276→6.798)
HEP-C++	0.000	0.994	-0.628	(2.621→6.890)
HEP-Java	0.073	0.994	-1.013	(2.426→7.310)

Table 5: Evaluation of CodeIP on four datasets. Correctness: pass@1; Detectability: AUROC; Imperceptibility: PPL change from non-watermarked to watermarked code.

Table 5 reports the reproduced results for CodeIP, a grammar-guided multi-bit watermarking method that employs a type prediction model trained on

CodeSearchNet for grammar-aware token selection during code generation. All experiments are based on the original implementation and evaluated on four benchmarks: MBPP+, HumanEval+, HumanEvalPack-C++, and HumanEvalPack-Java.

CodeIP maintains high detectability, achieving AUROC values above 0.94 across all datasets. This result indicates that grammar-based watermark insertion enables reliable message extraction. However, this detection capability is accompanied by substantial drops in correctness and imperceptibility. The pass@1 scores decrease sharply, approaching zero on HumanEval+ and HumanEvalPack-C++, which shows that syntactic validity enforced by the type predictor does not necessarily imply functional correctness. The large perplexity gap of up to +4.9 indicates that the interaction between watermarking and type-prediction constraints can shift the token distribution, leading to deviations from the model's natural generation pattern.

Weight (α, β, ζ)	MBPP+	HumanEval+	HEP-C++	HEP-Java
$(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$	0.270	0.296	0.122	0.018
$(0.5, 0.25, 0.25)$	0.241	0.274	0.095	0.041
$(0.25, 0.5, 0.25)$	0.381	0.361	0.267	0.145
$(0.25, 0.25, 0.5)$	0.325	0.331	0.174	0.084

Table 6: STEM scores of CodeIP under different (α, β, ζ) on MBPP+, HumanEval+, HumanEvalPack-C++, and HumanEvalPack-Java.

Table 6 presents the STEM scores of CodeIP under different weight settings. Across all configurations, detectability remains consistently high, while correctness- or imperceptibility-focused STEM scores are significantly lower than those of other methods. Even under weight settings emphasizing correctness or imperceptibility, CodeIP does not achieve balanced performance, showing limited robustness under varying evaluation preferences. These findings demonstrate that maximizing watermark detectability alone severely degrades code quality. In contrast, STONE embeds watermarks only within non-syntax tokens and achieves stable performance across correctness, detectability, and imperceptibility as captured by the STEM metric.

B Experimental Results on Llama-3.1-8B

Tables 7 and 8 summarize the experimental results obtained with the Llama-3.1-8B model on MBPP+ and HumanEval+. STONE achieves the highest STEM scores across all datasets and weight settings, demonstrating that the non-syntax watermarking strategy generalizes well to larger models. EWD shows slightly higher detectability on HumanEval+, which may be related to differences in the output distribution of Llama-3.1-8B under uniform insertion. In this setting, the cumulative bias introduced by EWD is more effectively reinforced across long and diverse sequences, leading to a small gain in detectability. Such an effect is not observed under Qwen2.5-Coder-7B, whose flatter distributions yield comparable results between EWD and SWEET. Overall, STONE preserves stronger correctness and achieves a more balanced trade-off, resulting in the highest overall STEM scores.

Across different weight configurations, STONE maintains stable performance even when correctness or imperceptibility is prioritized. The method remains competitive under detectability-oriented settings without functional degradation. This consistency confirms that syntax-aware watermarking effectively preserves code quality while sustaining robust watermark signals.

Method	MBPP+			HumanEval+		
	Correct.	Detect.	Imperc.	Correct.	Detect.	Imperc.
KGW	0.414	0.831	0.953	0.323	0.507	**0.989**
EWD	0.414	0.931	0.953	0.323	**0.755**	**0.989**
SWEET	0.404	0.927	**0.968**	0.354	0.711	0.978
STONE	**0.446**	**0.945**	0.954	**0.372**	0.741	0.983

Table 7: Experimental results on MBPP+ and HumanEval+ (Llama-3.1-8B): Correctness, Detectability, and Imperceptibility.

Weight (α, β, ζ)	MBPP+				HumanEval+			
	KGW	EWD	SWEET	STONE	KGW	EWD	SWEET	STONE
$(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$	0.733	0.766	0.766	**0.782**	0.606	0.689	0.681	**0.699**
$(0.5, 0.25, 0.25)$	0.653	0.678	0.676	**0.698**	0.535	0.598	0.599	**0.617**
$(0.25, 0.5, 0.25)$	0.757	0.807	0.806	**0.823**	0.582	0.706	0.688	**0.709**
$(0.25, 0.25, 0.5)$	0.788	0.813	0.817	**0.825**	0.702	0.764	0.755	**0.770**

Table 8: STEM scores on MBPP+ and HumanEval+ (Llama-3.1-8B) under different weight settings (α, β, ζ). The left column lists weights for correctness, detectability, and imperceptibility.

C Syntax Elements in Python, C++, and Java

Table 9 categorizes the syntactic elements into five groups: keywords, whitespace, types, delimiters, and operators. The selection of syntactic elements reflects the distinct characteristics of each programming language.

Category	Python	C++	Java
Keywords	True, False, None, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield	alignas, alignof, and, and_eq, asm, auto, bitand, bitor, break, case, catch, class, compl, concept, const, consteval, constexpr, constinit, const_cast, continue, co_await, co_return, co_yield, decltype, default, delete, do, dynamic_cast, else, enum, explicit, export, extern, false, for, friend, goto, if, inline, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, requires, return, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, using, virtual, volatile, while, xor, xor_eq, override	abstract, assert, break, case, catch, class, const, continue, default, do, else, enum, extends, final, finally, for, goto, if, implements, import, instanceof, interface, native, new, null, package, private, protected, public, return, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while, true, false
Whitespace	space, \n, \t	space, \n, \t	space, \n, \t
Types	int, float, complex, str, bytes, bool, list, tuple, set, dict, NoneType	int, float, double, bool, char, short, long, void, unsigned, signed, size_t, ptrdiff_t, wchar_t, char8_t, char16_t, char32_t	byte, short, int, long, float, double, boolean, char, String, Object
Delimiters	(,), [,], {, }, ‘,’, :, ‘.’, ;, @, ->, ...	(,), [,], {, }, ‘,’, :, ‘.’, ;, ->, ::, ...	(,), [,], {, }, ‘,’, :, ‘.’, ;, @, ->, ::, ...
Operators	+, -, *, /, %, **, //, =, ==, !=, >, <, >=, <=, +=, -=, *=, /=, %=, //=, **=, &, |, «, », ^, ~	+, -, *, /, %, ++, –, =, ==, !=, >, <, >=, <=, &&, ||, !, &, |, ^, ~, <<, >>, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, .*, ->*	+, -, *, /, %, ++, –, =, ==, !=, >, <, >=, <=, &&, ||, !, &, |, ^, ~, <<, >>, >>>, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=, >>>=

Table 9: Comparison of five categories of syntax elements across Python, C++, and Java.

Weight (α, β, ζ)	MBPP+				HumanEval+				HEP-C++				HEP-Java			
	KGW	EWD	SWEET	STONE	KGW	EWD	SWEET	STONE	KGW	EWD	SWEET	STONE	KGW	EWD	SWEET	STONE
$(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$	0.775	0.819	0.787	**0.848**	0.694	0.763	0.754	**0.781**	0.730	0.750	0.735	**0.780**	0.642	0.675	0.631	**0.715**
$(0.5, 0.25, 0.25)$	0.706	0.739	0.716	**0.778**	0.664	0.716	0.709	**0.732**	0.692	0.706	0.697	**0.741**	0.578	0.603	0.577	**0.648**
$(0.25, 0.5, 0.25)$	0.789	0.856	0.807	**0.881**	0.651	0.755	0.743	**0.780**	0.703	0.733	0.711	**0.768**	0.618	0.668	0.618	**0.716**
$(0.25, 0.25, 0.5)$	0.830	0.863	0.838	**0.883**	0.767	0.810	0.808	**0.830**	0.796	0.811	0.796	**0.833**	0.730	0.755	0.699	**0.781**

Table 10: STEM scores under different weight settings (α, β, ζ). The left column lists weights for correctness, detectability, and imperceptibility. Bold indicates the best score in each row.

D Comprehensive Evaluation across STEM Weight Configurations

Table 10 reports representative STEM variants using equal weights and three dimension-focused settings. We further explore all valid weight configurations where $\alpha, \beta, \zeta \in \{0.0, 0.1, \ldots, 1.0\}$ and $\alpha + \beta + \zeta = 1$, resulting in 66 combinations. For each configuration, we compute the weighted STEM score and record the method with the highest value. Across all datasets, STONE performs consistently well throughout the entire weight space, ranking first in 97.0% of configurations on MBPP+, 90.9% on HumanEval+, 98.5% on HumanEvalPack-C++, and 95.5% on HumanEvalPack-Java. This consistency indicates that the advantage of STONE does not depend on specific weight choices but generalizes across diverse evaluation preferences, demonstrating robust performance under varying priorities of correctness, detectability, and imperceptibility.

E Syntax Token Coverage in SWEET

Table 11 presents an analysis of token selection by SWEET using two measures: (1) the proportion of selected tokens exceeding the entropy threshold among all generated tokens, and (2) the proportion of syntax-related tokens among those selected. We use the optimal settings ($\gamma = 0.25$, $\delta = 3.0$, entropy threshold $= 0.9$) and evaluate threshold values in the range $\{0.7, 0.8, 0.9, 1.0, 1.1\}$. The results show an inverse relationship between the threshold and token coverage: lower thresholds select more tokens overall but include more syntax elements, whereas higher thresholds reduce syntax

Entropy Threshold	MBPP+		HumanEval+		HEP-C++		HEP-Java	
	Selected Tokens (%)	Syntax Tokens (%)	Selected Tokens (%)	Syntax Tokens (%)	Selected Tokens (%)	Syntax Tokens (%)	Selected Tokens (%)	Syntax Tokens (%)
0.7	26.03	12.82	35.28	12.84	30.10	12.99	26.60	11.98
0.8	21.40	12.59	32.67	12.24	25.93	12.35	22.47	11.06
0.9	**19.64**	**11.39**	**28.98**	**12.60**	**22.65**	**11.79**	**18.52**	**11.68**
1.0	16.59	11.39	24.84	11.83	18.96	10.56	16.13	11.15
1.1	14.94	10.30	23.22	11.81	15.71	9.82	14.39	10.91

Table 11: Token selection statistics of SWEET across four datasets. *Selected Tokens (%)* denotes the proportion of tokens exceeding the entropy threshold out of all tokens, and *Syntax Tokens (%)* represents the percentage of syntax-related tokens among those selected. Bold values indicate the optimal entropy threshold used in SWEET.

token inclusion. This pattern shows that SWEET effectively targets high-entropy regions but does not completely avoid syntax-related tokens.

Even under the optimal configuration, a notable portion of syntax tokens remains among those selected for watermarking. In the HumanEval+ dataset, syntax tokens account for 12.6% of all selected tokens, including delimiters (49.29%), whitespace (38.17%), keywords (9.44%), types (3.17%), and operators (2.78%). This indicates that delimiters and whitespace dominate syntax-related selections, which directly affects code execution.

Category	**Examples**
Keywords	def, or, import, for, True, assert, return, is, pass, None, False, in, not, if, from
Whitespace	space, \n
Types	int, set, str
Delimiters	. , : , {, }, [,], (,)
Operators	*, %, =, -, **, &, ^, /, +, |

Table 12: Examples of syntax tokens selected for watermarking in SWEET.

Table 12 lists representative examples. Delimiters such as colons (':') and commas (','), essential for code structure, can cause parsing errors when modified (e.g., omitting a colon in 'for i in range(10):' leads to a failure). Although keywords and operators appear less frequently, their modification can still alter program behavior, for example, replacing 'True' with 'False' or '+' with '-'. Minimizing the inclusion of such syntax-critical tokens can make syntax-aware watermarking more reliable and less disruptive to functionality.

F Comparison with Non-watermarked Code

We evaluate the functional impact of watermarking by comparing the correctness of code generated with and without watermark insertion. Non-watermarked code provides an empirical upper bound in our evaluation setting. STONE achieves scores nearly identical to this bound, showing that watermark insertion has minimal influence on code execution. Among all watermarking methods, STONE reduces correctness the least, which indicates that it embeds provenance information effectively while preserving the original program behavior.

Method	MBPP+	HumanEval+
No watermark	0.571	0.595
KGW	0.499	0.573
EWD	0.499	0.573
SWEET	0.502	0.574
STONE	**0.571**	**0.587**

Table 13: Correctness comparison between non-watermarked and watermarked code. Correctness is measured as pass@1 on MBPP+ and HumanEval+.