# Who Wrote this Code? Watermarking for Code Generation

Anonymous ACL submission


## Abstract

Since the remarkable generation performance of large language models raised ethical and legal concerns, approaches to detect machine-generated text by embedding watermarks are being developed. However, we discover that the existing works fail to function appropriately in code generation tasks due to the task's nature of having low entropy. Extending a logit-modifying watermark method, we propose **S**elective **W**at**E**rmarking via **E**ntropy **T**hresholding (SWEET), which enhances detection ability and mitigates code quality degeneration by removing low-entropy segments at generating and detecting watermarks. Our experiments show that SWEET significantly improves code quality preservation while outperforming all baselines, including post-hoc detection methods, in detecting machine-generated code text. Our code is available in the supplementary materials.


## 1 Introduction

In understanding and generating software programs, large language models have rapidly advanced towards expert-like proficiency (Chen et al., 2021; Luo et al., 2023; Li et al., 2023b; Nijkamp et al., 2023; Zheng et al., 2023; Gunasekar et al., 2023; Touvron et al., 2023; OpenAI, 2023a). This breakthrough in the automation of the coding process improves the productivity and efficiency of software engineer and lowers the barriers to creating programs for non-experts (Vaithilingam et al., 2022).

However, this advance comes with significant legal, ethical, and security concerns, including code licensing issues, code plagiarism, code vulnerability, and malware generation (He and Vechev, 2023; Sandoval et al., 2023; Pearce et al., 2022; Carlini et al., 2021; Mirsky et al., 2023; Hazell, 2023). For example, there is an ongoing class-action copyright lawsuit between a group of individuals and Microsoft, GitHub, and OpenAI, arising from allegations of unlawful utilization and reproduction of the source code[1][2]. Furthermore, shortly after the launch of ChatGPT, numerous malicious actors on the Dark Web were observed sharing machine-generated malware and spear phishing tutorials[3]. Therefore, the development of reliable tools for detecting machine-generated code is a very timely matter and is of utmost importance for fairly deploying LLMs with coding capabilities.

Despite the need for immediate treatment of the

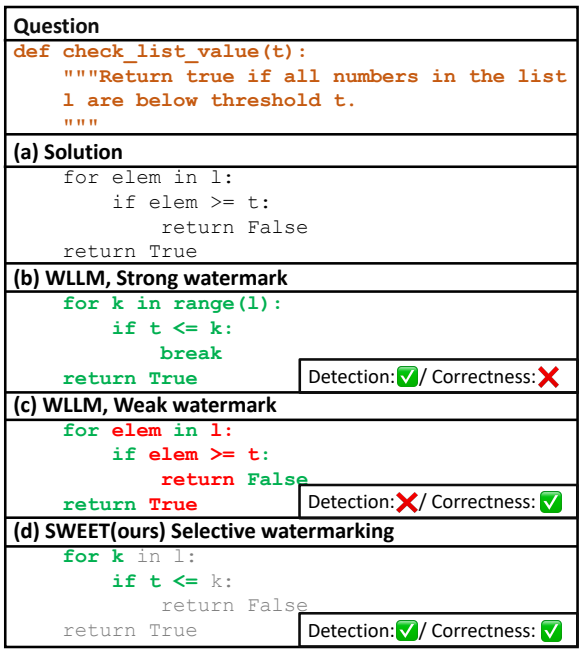


Figure 1: Illustrated comparison of WLLM (Kirchenbauer et al., 2023a) and SWEET (ours). Note that this example is a short hypothetical explanatory example. LLMs can generate working source code (a) without a watermark. Strong watermark (b) or weak watermark (c) may result in detection or correctness failure, but (d) selective watermarking may avoid both failures.

[1] Code plagiarism
[2] Code licensing issue
[3] Malware generation

machine-generated code detection problem, few efforts have been made to address it. Instead, many works still prioritize a detection problem on normal text (Solaiman et al., 2019; Ippolito et al., 2020; Guo et al., 2023; Tian and Cui, 2023; OpenAI, 2023b; Yu et al., 2023; Gehrmann et al., 2019; Mitchell et al., 2023; Yang et al., 2023). While these *post-hoc* detection methods (i.e., no control during the text generation) have demonstrated powerful performance in the many domain of natural language tasks, their application to programming language remains unexplored.

Contrary to the post-hoc detection methods, another line of research for detecting machine-generated text has gained attention: *Watermarking-based* methods, which embed a hidden signal within the generated text (Kirchenbauer et al., 2023a,b; Kuditipudi et al., 2023; Wang et al., 2023). For example, a method proposed in Kirchenbauer et al. (2023a) – which we refer to as WLLM (Watermarking for Large Language Models) – randomly divides the entire vocabulary into two groups (i.e., the green list and the red list) at each generation step and enhance the probability of green list tokens to be sampled. By adding scalar values to the logits of a green list tokens, the model favors generating tokens from the green list rather than the red one. To detect the watermark in a text, we count the number of green tokens and check whether this number is statistically significant (through hypothesis testing) to conclude whether the model output is generated without knowledge of the green-red rule.

While both watermarking-based methods and post-hoc detection methods work well in many language generation tasks, we observe that these performances do not transfer well to code generation tasks, for example, in Figure 1. In other words, it is much more challenging to embed watermarks in a detectable way without impairing the code functionality. We attribute this to the nature of extremely low entropy[4] of code generation. If watermarking is applied strongly, it can severely degrade the quality of the model output, which is particularly critical in code generation, as a single violation of a rule can break the entire code (see "strong watermark" in Figure 1). On the other hand, if watermarking is applied too weakly, the low entropy hinders properly embedding watermarks and results in insufficient green tokens appearing, leading to increased difficulty in detection (see "weak watermark" in Figure 1). These failures are not significant in plain text generation because the relatively higher entropy allows for more flexibility in candidate selections for watermarking.

To address these failure modes, we extend the WLLM and propose **S**elective **W**at**E**rmarking via **E**ntropy **T**hresholding (SWEET) for Code LLMs (and LLMs). Instead of applying the green-red rule to every single token during generation, we only apply the rule to tokens with *high enough entropy* given a threshold. That is, we do not apply the green-red rule to the important tokens for making functional code, while making sure there are enough green list tokens to make a detectable watermark for less important tokens, hence, directly addressing each of the above failure modes. In code generation tasks, our method outperforms all baselines, including post-hoc detection methods, in detecting machine-generated code while achieving less code quality degradation than WLLM. Furthermore, through various analyses, we demonstrate that our method operates well even without prompts or with a small surrogate model, indicating its robust performance under practical settings.

Our contributions are as follows:

- We are the first to empirically explore the breakdown of existing watermarking and post-hoc detection methods in the code domain.
- We propose a simple yet effective method called SWEET, which improves WLLM (Kirchenbauer et al., 2023a) and achieves significantly higher performance in machine-generated code detection while preserving code quality more than WLLM.
- We have demonstrated the practical applicability and predominance of our method even in real-world settings, i.e., 1) without prompts, 2) utilizing a smaller model as a detector, or 3) under paraphrasing attacks.

## 2 Related Work

**Software Watermarking** Software watermarking is the research field where a secret signal is embedded in the code without affecting its performance, to prevent software piracy. Static watermarking (Hamilton and Danicic, 2011; Li and Liu, 2010; Myles et al., 2005) imprints watermarks typically through code replacement and reordering.

[4]We calculate entropy over the probability of the next token prediction. Please refer to Eq. 5 for details.

On the other hands, dynamic watermarking (Wang et al., 2018; Ma et al., 2019) injects watermarks during the compiling or executing stage of a program. For a detailed survey, please refer to Dey et al. (2018).

Watermarking code text generated from a LLM is closer to static watermarking. For example, Li et al. (2023c) proposes a method employing the replacement of synonymous code. However, since this method heavily relies on language-specific rules, a malicious user knowing these rules could reverse the watermarking.

**LLM Text Watermarking** The majority of watermarking methods for texts from LLMs are based on the modification of the original text via a predefined set of rules (Atallah et al., 2001, 2002; Kim et al., 2003; Topkara et al., 2006; Jalil and Mirza, 2009; Meral et al., 2009; He et al., 2022a,b) or another language model, such as transformer-based networks. (Abdelnabi and Fritz, 2021; Yang et al., 2022; Yoo et al., 2023).

Recently, a line of work embeds watermarks into tokens during the sampling process of LLMs (Liu et al., 2024). They embed watermarks within LLM-generated texts by either motifying logits from the LLM (Kirchenbauer et al., 2023a,b; Liu et al., 2023a; Takezawa et al., 2023; Hu et al., 2023) or manipulating the sampling procedure (Christ et al., 2023; Kuditipudi et al., 2023). Moreover, some recent works focus on the robustness of watermarks against attacks to remove watermarks (Zhao et al., 2023; Liu et al., 2023b; Ren et al., 2023). Lastly, Gu et al. (2023) investigates the learnability of watermarks in the distillation process from teacher to student model.

However, these watermark methods exhibit vulnerability in their watermark detection performance under low entropy situations (Kirchenbauer et al., 2023a; Kuditipudi et al., 2023), and a limited number of studies, such as CTWL (Wang et al., 2023), try to handle it. We directly address the degradation of watermark detection performance in low entropy situations and demonstrate our method's efficacy in low entropy tasks, such as code generation.

**Post-hoc Detection** Post-hoc detection methods aim to differentiate between human-authored and machine-generated text without embedding any signal during generation. One line of work leverages perplexity-based features like GPTZero (Tian and Cui, 2023), Sniffer (Li et al., 2023a), and LLMDet (Wu et al., 2023). Another line of work uses pre-trained LM, such as RoBERTa (Liu et al., 2019), and fine-tunes it as a classifier to identify the source of text (Solaiman et al., 2019; Ippolito et al., 2020; OpenAI, 2023b; Guo et al., 2023; Yu et al., 2023; Mitrović et al., 2023). Meanwhile, some recent works tackle the detection problem without additional training procedures, such as GLTR (Gehrmann et al., 2019), DetectGPT (Mitchell et al., 2023), and DNA-GPT (Yang et al., 2023). However, post-hoc detection methods remain challenging. For example, while the GPTZero (Tian and Cui, 2023) is still in service, OpenAI's AI text classifier (OpenAI, 2023b) was discontinued after six months due to low accuracy rates. Furthermore, we have demonstrated that post-hoc detection methods failed to detect machine-generated code, with low entropy.

## 3 Method

We propose a new watermarking method, SWEET, that selectively watermarks tokens only with high enough entropy.

### 3.1 Motivation

Although the previous watermarking method WLLM (Kirchenbauer et al., 2023a) can be applied to any domain of LLM-generated text[5], it incurs two critical problems during embedding and detecting watermarks in code generation, attributed to a dilemma regarding watermark strength.

**Watermarking causes performance degradation.** There are only a few different ways of expressing the same meaning in a programming language, and just one wrong token can be attributed to undesirable outputs. If watermarks are embedded strongly, as WLLM randomly divides the vocabulary into green and red lists without leveraging any information about the context, promoting the logits of only green list tokens must heighten the chance of generating the wrong token. For example, in Figure 2 (a), after "return" token in the second row, the next token with the highest logit is "sum", which is also part of the canonical solution. However, WLLM puts "sum" into the red list while putting "mean" into the green list. Hence, the sampled token was "mean", resulting in a syntax error.

**Low Entropy Sequences Avoid Being Watermarked.** Another critical issue is when watermark strength is too weak to embed watermarks into a text with low entropy. If a red list token has a too

[5]Please refer to Appendix A for the details of WLLM.

**Question (HumanEval[4])**

```
from typing import List


def mean_absolute_deviation(numbers: List[float]) -> float:
    """ For a given list of input numbers, calculate Mean Absolute Deviation
    around the mean of this dataset.
    Mean Absolute Deviation is the average absolute difference between each
    element and a centerpoint (mean in this case):
    MAD = average | x - x_mean |
    >>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
    1.0
    """
```

**Canonical solution**

```
    mean = sum(numbers) / len(numbers)
    return sum(abs(x - mean) for x in numbers) / len(numbers)
```

**(a) WLLM**

```
    centerpoint = sum(numbers) / float(len(numbers))
    return mean([abs(number - centerpoint) for number in numbers])
```

Correctness: ✗ / z-score: 2.45 / watermarking ratio: 1.0

**(b) SWEET – Entropy Threshold (Low)**

```
    centerpoint = sum(numbers) / float(len(numbers))
    return mean([abs(number - centerpoint) for number in numbers])
```

Correctness: ✗ / z-score: 3.39 / watermarking ratio: 0.44

**(c) SWEET – Entropy Threshold (Moderate)**

```
    centerpoint = sum(numbers) / len(numbers)
    distances = []
    for number in numbers:
        distance = abs(number - centerpoint)
        distances.append(distance)
    m_a_d = sum(distances) / len(distances)
    return m_a_d
```

Correctness: ✓ / z-score: 4.96 / watermarking ratio: 0.20

**(d) SWEET – Entropy Threshold (High)**

```
    centerpoint = sum(numbers) / len(numbers)
    distances = []
    for number in numbers:
        distance = abs(number - centerpoint)
        distances.append(distance)
    m_a_d = sum(distances) / len(distances)
    return m_a_d
```

Correctness: ✓ / z-score: 4.67 / watermarking ratio: 0.19

Figure 2: A real example of HumanEval/4 for comparing between (a) WLLM and (b)–(d) our SWEET with different thresholds. Text colors annotate whether tokens are in the green or red list. Gray tokens have entropy smaller than the threshold and are not watermarked. The intensity of the yellow background color visualizes the entropy value. (a) While WLLM produces an incorrect code and less detectable watermarks with a few green tokens (low z-score), (b)-(d) SWEET improves both code quality and z-score by selectively embedding and detecting watermarks using an entropy threshold. Interestingly, (c) the z-score peaks with a moderate threshold, and (d) as the threshold increases, the z-score declines due to the decrease in the watermarking ratio.

high logit value to be inevitably generated, it hinders watermark detection. For example, in Figure 2 (a), tokens with white backgrounds representing low entropy have few green tokens. This becomes much more fatal in code generation tasks where outcomes are relatively shorter than the plain text, such as asking only a code block of a function[6]. The WLLM detection method is based on a statistical test, which involves counting the number of green list tokens in the entire length. However detecting watermarks based on a statistical test deteriorates if the length is short.[7]

### 3.2 The SWEET Method

SWEET can mitigate this dilemma regarding the watermark strength by distinguishing watermark-applicable tokens, meaning we embed and detect watermarks only within tokens with high entropy.

**Generation.** The generation step of our method is in Algorithm 1. Given a tokenized prompt $\boldsymbol{x} = \{x_0, \ldots, x_{M-1}\}$ and already generated tokens $\boldsymbol{y}_{[:t]} = \{y_0, \ldots, y_{t-1}\}$, a model calculates an entropy value ($H_t$) of the probability distribution for $y_t$. We then only apply the watermarking when $H_t$ is higher than the threshold, $\tau$. We randomly bin a vocabulary by green and red with a fixed green token ratio $\gamma$. If a token is selected to be watermarked, we add a constant $\delta$ to green tokens' logits, aiming to promote the sampling of the green tokens. By limiting the promotion of green tokens only to tokens with high entropy, we prevent the model's logit distribution changes for tokens where the model has confidence (and, therefore, low entropy), resulting in preserving code quality.

**Detection.** We outline our detection process in Algorithm 2. Given a token sequence $\boldsymbol{y} = \{y_0, \ldots, y_{N-1}\}$, our task is to detect watermarks within $\boldsymbol{y}$; therefore, determine whether it is generated from the specific language model. Like in the generation phase, we compute the entropy values $H_t$ for each $y_t$. Let $N^h$ denote the number of tokens that have an entropy value $H_t$ higher than the threshold $\tau$, and let $N_G^h$ denote the number of green tokens among in $N^h$. Finally, with the green list ratio among entire vocabulary $\gamma$ used in the generation step, we compute a $z$-score under the null hypothesis where the text is not watermarked by

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

We can say the text is watermarked more confidently as $z$-score goes higher. We set $z_{\text{threshold}}$ as a cut-off score. If $z > z_{\text{threshold}}$ holds, we decide that

[6]The average token length of human-written solution codes in HumanEval, MBPP, and DS-1000 datasets is only 57.

[7]We measured detectability according to the length of generated texts and observed that WLLM performs relatively poorly while SWEET is robust in detecting watermarks within short texts. For more details, please refer to Appendix E.

the watermark is embedded in $\boldsymbol{y}$ and thus generated by the LLM. The effect of the entropy threshold in the detection phase is described in the following section.

### 3.3 Effect of Entropy Thresholding

This section shows that selective watermark detection based on the entropy threshold improves the detectability.

Theorem 1 implies that we can ensure a higher lower bound of $z$-score by the SWEET detection method than WLLM. Recalling Sec 3.1, this is achieved by ignoring tokens with low entropy, leading to increases in the ratio of green tokens within the text and detectability.

For the sake of theoretical analysis, we use *spike entropy* (Eq. 4), which is a variant of entropy defined in Kirchenbauer et al. (2023a). In practice, we use the entropy in Eq. 5.

**Theorem 1.** *Consider a token sequence $\boldsymbol{y} = \{y_0, \ldots, y_{N-1}\}$ generated by a watermarked code LLM. $(S_0, \ldots, S_{N-1})$ is a sequence of corresponding spike entropy, in which the modulus is $\frac{(1-\gamma)(e^\delta-1)}{1+(e^\delta-1)\gamma}$. Let $\tau$ be an entropy threshold, $N^l$ and $N^h$ be the number of tokens whose spike entropy is lower or higher than the threshold.*

*If the following assumption regarding the ratio of low entropy tokens holds*

$$\frac{N^l}{N} \leq 1 - \left(\frac{\alpha\overline{S} - 1}{\alpha\overline{S^h} - 1}\right)^2$$

*then there is a lower bound of $z$-score that is always higher when the entropy threshold is applied, where $\alpha = \frac{e^\delta}{1+(e^\delta-1)\gamma}$, $\overline{S} = \Sigma_{t=1}^{N} S_t/N$, and $\overline{S^h} = \Sigma_{t=1}^{N} S_t \times \mathbb{1}(S_t \geq \tau)/N^h$.*

*Remark.* The assumption means choosing an entropy threshold that does not ignore too many tokens ($N^l$) is important.

## 4 Experiments

We conduct a series of experiments to evaluate the effectiveness of our watermarking method in code generation for two aspects: (i) quality preserving ability and (ii) detection strength. Our base model is StarCoder (Li et al., 2023b), which is an open-source LLM specifically for code generation. We also conduct experiments on one of the general-purpose LLM, LLaMA2 (Touvron et al., 2023) (see the results in Appendix F).

### 4.1 Tasks and Metrics

We select three python code generation tasks, HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and DS-1000 (Lai et al., 2023), as our testbeds. These tasks contain python programming problems, test cases, and human-written canonical answers. Language model is prompted with programming problems and expected to generate the correct code that can pass the test cases.

To evaluate the functional quality of generated source code, we use pass@k (Chen et al., 2021) by generating $n(> k)$ outputs for each programming problems. This metric estimates the percentage of code generated correctly-performing. For the detection ability, we use AUROC (i.e., Area Under ROC) value as a main metric. We also report the true positive rate (TPR; correctly detecting LLM-generated code as LLM-generated) when the false positive rate (FPR; falsely detecting human-written code as LLM-generated) is confined to be lower than 5%. This is to observe the detection ratio of a practical setting, where high false positive is more undesirable than false negative.

### 4.2 Baselines

We compare SWEET with machine-generated text detection baselines. *Post-hoc* detection baselines do not need any modification during generation so that they never impair the quality of the model output. LOGP(X), LOGRANK (Gehrmann et al., 2019), and DETECTGPT (Mitchell et al., 2023) are zero-shot detection methods that need no labeled datasets. GPTZERO (Tian and Cui, 2023) and OPENAI CLASSIFIER (Solaiman et al., 2019) are trained classifiers. For *Watermarking-based* methods, we have included two baselines: WLLM (Kirchenbauer et al., 2023a) and EXP-EDIT (Kuditipudi et al., 2023). To embed a watermark, methods that distort the model's sampling distribution, such as WLLM or ours, tend to have better detection ability, but degradation of text quality may arise. On the other hand, EXP-EDIT is expected to cause no degradation in text quality as they do not distort the sampling distribution of the model.[8] More details of implementation are in Appendix D.

[8]When evaluating code generation performance through pass@1, a low temperature was applied to all models. However, the spiky distribution resulting from the low temperature hindered EXP-EDIT from adequately embedding watermarking. Therefore, we have also included EXP-EDIT baseline with a high entropy by setting temperature=1.0 and top-p=1.0.

| Method | | HumanEval | | | | MBPP | | | | DS-1000 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | PASS@1 | AUROC | TPR | FPR | pass@1 | AUROC | TPR | FPR | pass@1 | AUROC | TPR | FPR |
| **Non-watermarked** | | **33.4** | - | - | - | **37.8** | - | - | - | **26.3** | - | - | - |
| **Non-watermarked (w/ high entropy)** | | 18.3 | - | - | - | 21.4 | - | - | - | 12.7 | - | - | - |
| **Post-hoc** | LOG P(X) | **33.4** | 0.533 | 0.113 | < 0.05 | **37.8** | 0.525 | 0.054 | < 0.05 | **26.3** | 0.566 | 0.100 | < 0.05 |
| | LOGRANK | | 0.553 | 0.127 | < 0.05 | | 0.527 | 0.052 | < 0.05 | | 0.562 | 0.105 | < 0.05 |
| | DETECTGPT (T5-3B) | | 0.549 | 0.092 | < 0.05 | | 0.531 | 0.040 | < 0.05 | | 0.433 | 0.070 | < 0.05 |
| | DETECTGPT | | 0.533 | 0.165 | < 0.05 | | 0.565 | 0.158 | < 0.05 | | 0.606 | 0.113 | < 0.05 |
| | GPTZERO | | 0.521 | 0.122 | < 0.05 | | 0.449 | 0.026 | < 0.05 | | 0.539 | 0.063 | < 0.05 |
| | OPENAI CLASSIFIER | | 0.518 | 0.053 | < 0.05 | | 0.500 | 0.036 | < 0.05 | | 0.524 | 0.075 | < 0.05 |
| **Watermarking** | EXP-EDIT | **33.6** | 0.489 | 0.085 | < 0.05 | **37.5** | 0.536 | 0.044 | < 0.05 | **26.2** | 0.546 | 0.066 | < 0.05 |
| | EXP-EDIT (w/ high entropy) | 19.3 | 0.733 | 0.427 | < 0.05 | 22.7 | 0.744 | 0.33 | < 0.05 | 12.7 | 0.743 | 0.378 | < 0.05 |
| | WLLM ($\Delta$PASS@1 $\sim -10\%$)$^\star$ | 29.6 | 0.822 | 0.402 | < 0.05 | 34.5 | 0.718 | 0.178 | < 0.05 | 23.9 | 0.627 | 0.152 | < 0.05 |
| | SWEET ($\Delta$PASS@1 $\sim -10\%$)$^\star$ | 32.6 | **0.943** | **0.835** | < 0.05 | 33.8 | **0.873** | **0.590** | < 0.05 | 23.7 | **0.815** | **0.384** | < 0.05 |
| | WLLM (AUROC$\geq$ 0.9)$^\dagger$ | 25.3 | 0.904 | 0.652 | < 0.05 | 24.2 | 0.930 | 0.718 | < 0.05 | 8.6 | 0.944 | 0.793 | < 0.05 |
| | SWEET (AUROC$\geq$ 0.9)$^\dagger$ | 32.6 | 0.943 | 0.835 | < 0.05 | 33.2 | 0.906 | 0.548 | < 0.05 | 18.8 | 0.924 | 0.649 | < 0.05 |

Table 1: **Main results** of code generation performance and detection ability. Since calibration on watermarking strength leads to trade-offs between code generation quality and detection ability, we present two results for WLLM and SWEET. $\star$ for the best detection score (i.e., AUROC and TPR) while allowing a code generation quality decrease of $\sim$10% compared to Non-watermarked, and $\dagger$ for the best code generation quality (PASS@1) among AUROC $\geq$ 0.9. The selected points are shown in Figure 3. We add EXP-EDIT and a Non-watermarked baseline with a high entropy setting (i.e., temperature=1.0 and top-p=1.0) since EXP-EDIT hardly detects watermarking in low entropy environments.

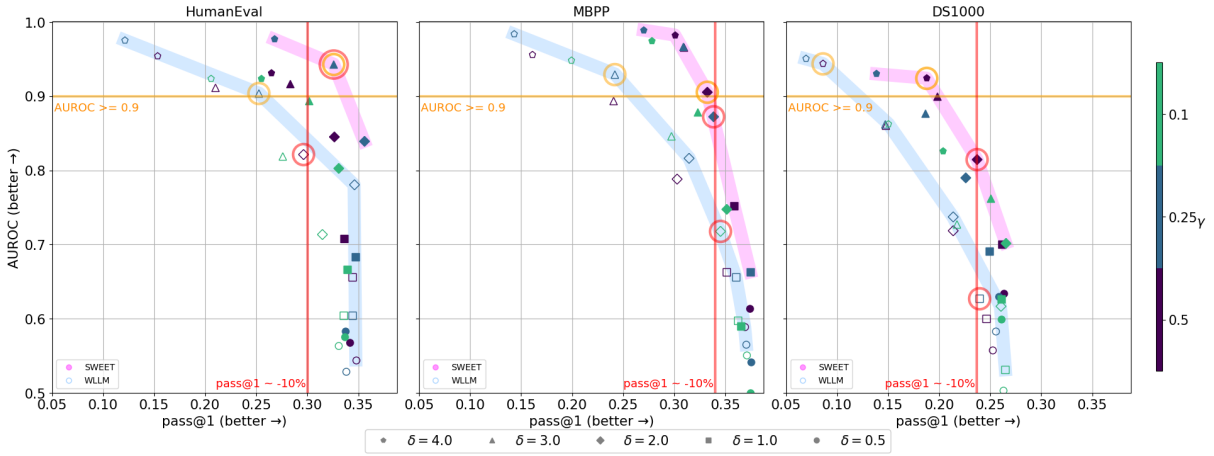


Figure 3: The tradeoff between AUROC and pass@1 of detecting real and generated samples of HumanEval, MBPP, and DS-1000 datasets. The pink line represents a Pareto frontier of SWEET, while the blue line represents that of WLLM. SWEET shows consistent dominance. The red/orange line and circles are the points used in Table 1. The entropy threshold value used for SWEET is 1.2 here, and Pareto frontier figures for all threshold values are in Figure 6.

## 5 Results

### 5.1 Main Results

Table 1 presents results from all baselines and our approach. In WLLM and SWEET, there is a clear trade-off between detection and code generation ability depending on the watermarking strength. Therefore, we measure the maximum scores of one domain while setting a lower bound for the scores of other domain. Specifically, to measure AUROC scores, we find the best AUROC scores around 90% of the pass@1 performance of the non-watermarked base model. On the other hand, for measuring pass@1, we select from those with an AUROC of 0.9 or higher.

**Detection Performance**. Table 1 shows that overall, our SWEET method outperforms all baselines in detecting machine-generated code with a price of 10% degradation of code functionality. Both in the MBPP and DS-1000 datasets, SWEET achieves AUROC of 0.873 and 0.815, respectively, whereas none of the baselines exceeded 0.8. SWEET even achieves an AUROC of 0.943 in HumanEval with a 2.4% degradation of code functionality. However, when only near 10% degradation of code functionality is allowed,

WLLM shows lower detection performance than our method. In the case of the distortion-free watermarking method, due to the lower entropy of the code generation task, EXP-EDIT fails to achieve an AUROC score exceeding 0.6 in all cases, and even EXP-EDIT with high entropy setting could not outperform our methods with regard of the detection performance. While all post-hoc detection baselines preserve code functionality as they do not modify generated code, none of them achieve an AUROC score above 0.6.[9]

**Code Quality Preservation**. In the last two rows of Table 1, despite the inevitable text quality degradation caused by WLLM and SWEET, our SWEET method preserves code functionality much more while maintaining the high detection ability of AUROC $> 0.9$ when compared to WLLM. Specifically, pass@1 of WLLM for HumanEval decreases from 33.4 to 25.3, a 24.3% loss in the code execution pass rate. Similarly, for the MBPP and the DS-1000 dataset, the drops in performances are 36.0% and 67.3%, respectively. On the other hand, our approach loses only 2.4% (HumanEval), 12.2% (MBPP), and 28.5% (DS-1000), respectively, which are significantly less than those of WLLM.

### 5.2 Comparison of Pareto Frontiers between SWEET and WLLM

In the cases of SWEET and WLLM, watermarking strength and spans can vary depending on the ratio of the green list tokens $\gamma$ and the logit increase value $\delta$. To demonstrate that SWEET consistently outperforms the baseline WLLM regardless of the values of $\gamma$ and $\delta$, we draw Pareto frontier curves with axes pass@1 and AUROC in Figure 3. We observe that the Pareto frontiers of SWEET are ahead of those of WLLM in all three tasks. Moreover, as presented in Figure 6, whatever value our approach chooses for the entropy threshold, SWEET outperforms the baseline in all configurations. This indicates that in a wide range of hyperparameter settings, our SWEET model can generate better results in terms of detection and code generation ability. Full results and different settings are in Appendix F.

[9]We defer a more in-depth discussion about the breakdown of Post-hoc methods to Appendix J.

## 6 Analysis

### 6.1 Impact of Entropy Thresholds

Figure 4 presents how code generation performance and detecting ability trade-off when calibrating the entropy threshold in our method. WLLM is when the entropy threshold is not applied (i.e., entropy threshold=0). As the entropy threshold increases, the ratio of watermarked tokens decreases, so the code generation performance converges to a non-watermarked base model. This indicates that our method always lies between the WLLM and a non-watermarked base model in terms of code generation performance. On the other hand, the detection ability, as the entropy threshold increases, reaches a local maximum but eventually declines. While our method with a moderate threshold effectively restricts generating the red list tokens compared to the WLLM, detection ability eventually decreases if the threshold is so high that few tokens are watermarked.

### 6.2 Detection Ability without Prompts

As entropy information is required in the detection phase, approximating entropy values for each generation time step $t$ is essential in our method. In the main experiments, we prepend the prompt used in the generation phase (e.g., the question of Fig. 2) before the target code to reproduce the same entropy. However, we hardly know the prompt used for a given target code in the real world. Thus, instead of using the *gold* prompt, we attach a common and general prompt for code generation to approximate the entropy information, such as `"def solution(*args): """Generate a solution"""`. We use five general prompts (see Appendix G), and their z-scores are averaged for use in detection.

Figure 7 demonstrates how the detection ability varies when using general prompts in the HumanEval dataset. SWEET with general prompts shows lower AUROC values than the original SWEET, indicating inaccurately approximated entropy information impairs detection ability. Nevertheless, it still outperforms the WLLM baseline regarding detection ability, drawing a Pareto frontier ahead of WLLM in all entropy threshold values.

### 6.3 Use of Surrogate Model

When detecting watermarks in a text, utilizing a smaller LM as a surrogate could be more computationally efficient and cost-effective (Wang et al.,

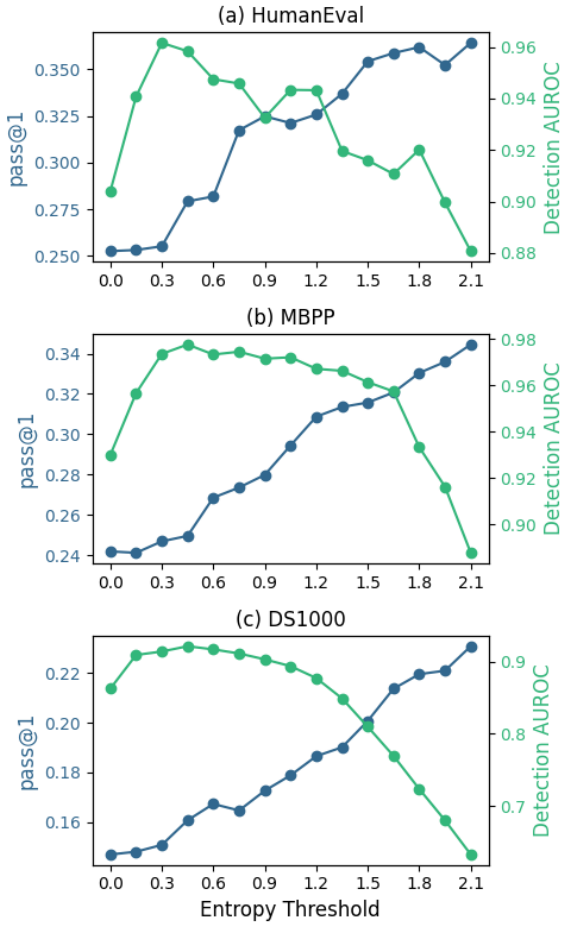


Figure 4: Plots of code quality pass@1 and detection AUROC when calibrating the entropy threshold of our methods, SWEET, on the three code benchmarks. We set $\gamma = 0.25$ and $\delta = 3.0$. While code generation performance increases with a higher entropy threshold, detection AUROC scores make an up-and-down curve.

2023). We investigate the impact of employing this surrogate model during the detection phase. Specifically, we generate watermarked code using the original model (LLaMA2-13B) and detect watermarks using a smaller model (LLaMA2-7B).

In the results of Figure 9, the detection performance declines are insignificant, and our approach utilizing the surrogate model continues to surpass the baseline. Such performance preservation may be due to that LLaMA2 7B and 13B are trained on the identical training corpus (Touvron et al., 2023). Further analysis for computational cost can be found in Appendix H.

### 6.4 Robustness to Variable Renaming

Even with the text watermarked, a malicious user might attempt to remove watermarks in the text by paraphrasing (Krishna et al., 2023; Sadasivan et al., 2023). Paraphrasing the code text is more restrictive than dealing with plain text because it must avoid triggering any code malfunctions. We assess the robustness of watermarking methods against paraphrasing by employing a straightforward approach - changing the names of variables. Specifically, we select variables in the watermarked code and rename them with randomly generated strings of varying lengths, ranging from 2 to 5 characters.

Figure 10 presents the results of the detection performance on the renamed variables in the code. All watermarking methods show the decline of AUROC scores when more variables are renamed, while our approaches continue to show better performances than baselines. However, our approaches also show that the AUROC scores drop to about 0.8 when all variables are renamed. We found that this is because variable names comprise a large proportion of high entropy tokens in the code text (See Appendix I for details).

## 7 Conclusion

We identified and emphasized the need for Code LLM watermarking, and formalized it for the first time. Despite the rapid advance of coding capability of LLMs, the necessary measures to encourage the safe usage of code generation models have not been implemented yet. Our experiments showed that existing watermarking and detection techniques failed to properly operate under the code generation setting. The failure occurred in two modes: either 1) the code does not watermark properly (hence, cannot be detected), or 2) the watermarked code failed to properly execute (degradation of quality). Our proposed method SWEET, on the other hand, improved both of these failure modes to a certain extent by introducing selective entropy thresholding which filters tokens that are least relevant to execution quality. In code generation tasks, our method performs better than baselines, including post-hoc detection methods, while achieving less code quality degradation. Moreover, comprehensive analysis demonstrates that our method still works well in real-world settings, specifically when the prompts are not given, utilizing even a smaller surrogate model, or under paraphrasing attacks.

## Limitations

We identify limitations of this work and suggest ways to mitigate them. We want to note that these

limitations are not *weaknesses* of our work as they are shared by the status quo of this field.

The first one is about the robustness issue. As users can tailor LLM's code to their specific needs, it is crucial to be robust against any kind of paraphrasing attacks. Though such robustness is not the focus of this work, we acknowledge the importance of robustness and leave it for future work.

The necessity to calibrate an additional hyperparameter, the entropy threshold, for our method could pose another limitation. While we demonstrate that our method with a moderate threshold value outperforms the baselines (see Sec 6.1), the imperative manual procedure of selecting the entropy threshold demands additional computational expenses.

Furthermore, during detection, we also need the source Code LLM, hence this method works only in a completely white-box setting. Although it has been shown that employing even a smaller surrogate LM can still maintain the detection performances to some degree (see Sec 6.3), this can be a computational burden for some users who want to apply our work.

## Ethical Statement

Although watermarking methods are designed to address all potential misuse of LLMs by detecting machine-generated texts, they can simultaneously pose a new risk. For example, if a watermarking mechanism for a specific LLM is leaked to the public, a malicious user aware of this mechanism could abuse the watermarks to create unethical texts embedded with the model's watermarks. To prevent such scenarios, we recommend that all users exercise caution to avoid exposing the detailed mechanism, such as the key value for the hash function used to divide green and red lists in our method.

## References


Sahar Abdelnabi and Mario Fritz. 2021. Adversarial watermarking transformer: Towards tracing text provenance with data hiding. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 121–140. IEEE, IEEE.

Mikhail J. Atallah, Victor Raskin, Michael Crogan, Christian Hempelmann, Florian Kerschbaum, Dina Mohamed, and Sanket Naik. 2001. Natural language watermarking: Design, analysis, and a proof-of-concept implementation. In *Information Hiding*, pages 185–200. Springer, Springer Berlin Heidelberg.

Mikhail J. Atallah, Victor Raskin, Christian F. Hempelmann, Mercan Karahan, Radu Sion, Umut Topkara, and Katrina E. Triezenberg. 2002. Natural language watermarking and tamperproofing. In *Information Hiding*, pages 196–212. Springer, Springer Berlin Heidelberg.

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*.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *USENIX Security Symposium*, volume 6.

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*.

Miranda Christ, Sam Gunn, and Or Zamir. 2023. Undetectable watermarks for language models. *arXiv preprint arXiv:2306.09194*.

Ayan Dey, Sukriti Bhattacharya, and Nabendu Chaki. 2018. Software watermarking: Progress and challenges. *INAE Letters*, 4(1):65–75.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander Rush. 2019. GLTR: Statistical detection and visualization of generated text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 111–116, Florence, Italy. Association for Computational Linguistics.

Chenchen Gu, Xiang Lisa Li, Percy Liang, and Tatsunori Hashimoto. 2023. On the learnability of watermarks for language models. *arXiv preprint arXiv:2312.04469*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Biyang Guo, Xin Zhang, Ziyuan Wang, Minqi Jiang, Jinran Nie, Yuxuan Ding, Jianwei Yue, and Yupeng Wu. 2023. How close is chatgpt to human experts? comparison corpus, evaluation, and detection. *arXiv preprint arXiv:2301.07597*.

James Hamilton and Sebastian Danicic. 2011. A survey of static software watermarking. In *2011 World Congress on Internet Security (WorldCIS-2011)*, pages 100–107. IEEE, IEEE.

Julian Hazell. 2023. Large language models can be used to effectively scale spear phishing campaigns. *arXiv preprint arXiv:2305.06972*.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. *arXiv preprint arXiv:2302.05319*.

Xuanli He, Qiongkai Xu, Lingjuan Lyu, Fangzhao Wu, and Chenguang Wang. 2022a. Protecting intellectual property of language generation apis with lexical watermark. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10758–10766.

Xuanli He, Qiongkai Xu, Yi Zeng, Lingjuan Lyu, Fangzhao Wu, Jiwei Li, and Ruoxi Jia. 2022b. Cater: Intellectual property protection on text generation apis via conditional watermarks. *Advances in Neural Information Processing Systems*, 35:5431–5445.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *International Conference on Learning Representations*.

Zhengmian Hu, Lichang Chen, Xidong Wu, Yihan Wu, Hongyang Zhang, and Heng Huang. 2023. Unbiased watermark for large language models. *arXiv preprint arXiv:2310.10669*.

Daphne Ippolito, Daniel Duckworth, Chris Callison-Burch, and Douglas Eck. 2020. Automatic detection of generated text is easiest when humans are fooled. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1808–1822, Online. Association for Computational Linguistics.

Zunera Jalil and Anwar M. Mirza. 2009. A review of digital watermarking techniques for text documents. In *2009 International Conference on Information and Multimedia Technology*, pages 230–234. IEEE, IEEE.

Young-Won Kim, Kyung-Ae Moon, and Il-Seok Oh. 2003. A text watermarking algorithm based on word classification and inter-word space statistics. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 775–779. Citeseer, IEEE Comput. Soc.

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

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Manli Shu, Khalid Saifullah, Kezhi Kong, Kasun Fernando, Aniruddha Saha, Micah Goldblum, and Tom Goldstein. 2023b. On the reliability of watermarks for large language models. *arXiv preprint arXiv:2306.04634*.

Kalpesh Krishna, Yixiao Song, Marzena Karpinska, John Wieting, and Mohit Iyyer. 2023. Paraphrasing evades detectors of ai-generated text, but retrieval is an effective defense. *arXiv preprint arXiv:2303.13408*.

Rohith Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. 2023. Robust distortion-free watermarks for language models. *arXiv preprint arXiv:2307.15593*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Jun Li and Quan Liu. 2010. Design of a software watermarking algorithm based on register allocation. In *2010 2nd International Conference on E-business and Information System Security*, pages 1–4. IEEE, IEEE.

Linyang Li, Pengyu Wang, Ke Ren, Tianxiang Sun, and Xipeng Qiu. 2023a. Origin tracing and detecting of llms. *arXiv preprint arXiv:2304.14072*.

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

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

Aiwei Liu, Leyi Pan, Xuming Hu, Shu'ang Li, Lijie Wen, Irwin King, and Philip S. Yu. 2023a. An unforgeable publicly verifiable watermark for large language models.

Aiwei Liu, Leyi Pan, Xuming Hu, Shiao Meng, and Lijie Wen. 2023b. A semantic invariant robust watermark for large language models. *arXiv preprint arXiv:2310.06356*.

Aiwei Liu, Leyi Pan, Yijian Lu, Jingjing Li, Xuming Hu, Lijie Wen, Irwin King, and Philip S. Yu. 2024. A survey of text watermarking in the era of large language models.

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 preprint arXiv:1907.11692*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Haoyu Ma, Chunfu Jia, Shijia Li, Wantong Zheng, and Dinghao Wu. 2019. Xmark: Dynamic software watermarking using collatz conjecture. *IEEE Transactions on Information Forensics and Security*, 14(11):2859–2874.

Hasan Mesut Meral, Bülent Sankur, A. Sumru Özsoy, Tunga Güngör, and Emre Sevinç. 2009. Natural language watermarking via morphosyntactic alterations. *Computer Speech & Language*, 23(1):107–125.

Yisroel Mirsky, Ambra Demontis, Jaidip Kotak, Ram Shankar, Deng Gelei, Liu Yang, Xiangyu Zhang, Maura Pintor, Wenke Lee, Yuval Elovici, and Battista Biggio. 2023. The threat of offensive AI to organizations. *Computers & Security*, 124:103006.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. *The Fortieth International Conference on Machine Learning*.

Sandra Mitrović, Davide Andreoletti, and Omran Ayoub. 2023. Chatgpt or human? detect and explain. explaining decisions of machine learning model for detecting short chatgpt-generated text.

Ginger Myles, Christian Collberg, Zachary Heidepriem, and Armand Navabi. 2005. The evaluation of two software watermarking algorithms. *Software: Practice and Experience*, 35(10):923–938.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

OpenAI. 2023a. Gpt-4 technical report. *OpenAI Blog*.

OpenAI. 2023b. New ai classifier for indicating ai-written text. *OpenAI Blog*.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of GitHub copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, IEEE.

Jie Ren, Han Xu, Yiding Liu, Yingqian Cui, Shuaiqiang Wang, Dawei Yin, and Jiliang Tang. 2023. A robust semantics-based watermark for large language model against paraphrasing. *arXiv preprint arXiv:2311.08721*.

Vinu Sankar Sadasivan, Aounon Kumar, Sriram Balasubramanian, Wenxiao Wang, and Soheil Feizi. 2023. Can ai-generated text be reliably detected? *arXiv preprint arXiv:2303.11156*.

Gustavo Sandoval, Hammond A. Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*.

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*.

Yuki Takezawa, Ryoma Sato, Han Bao, Kenta Niwa, and Makoto Yamada. 2023. Necessary and sufficient watermark for large language models. *arXiv preprint arXiv:2310.00833*.

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

Umut Topkara, Mercan Topkara, and Mikhail J. Atallah. 2006. The hiding virtues of ambiguity. In *Proceedings of the 8th workshop on Multimedia and security*, pages 164–174. ACM.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7.

Lean Wang, Wenkai Yang, Deli Chen, Hao Zhou, Yankai Lin, Fandong Meng, Jie Zhou, and Xu Sun. 2023. Towards codable text watermarking for large language models. *arXiv preprint arXiv:2307.15992*.

Yilong Wang, Daofu Gong, Bin Lu, Fei Xiang, and Fenlin Liu. 2018. Exception handling-based dynamic software watermarking. *IEEE Access*, 6:8882–8889.

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

In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 2113–2133, Singapore. Association for Computational Linguistics.

Xi Yang, Jie Zhang, Kejiang Chen, Weiming Zhang, Zehua Ma, Feng Wang, and Nenghai Yu. 2022. Tracing text provenance via context-aware lexical substitution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 11613–11621. Association for the Advancement of Artificial Intelligence (AAAI).

Xianjun Yang, Wei Cheng, Linda Petzold, William Yang Wang, and Haifeng Chen. 2023. Dna-gpt: Divergent n-gram analysis for training-free detection of gpt-generated text. *arXiv preprint arXiv:2305.17359*.

KiYoon Yoo, Wonhyuk Ahn, Jiho Jang, and Nojun Kwak. 2023. Robust natural language watermarking through invariant features. In *Proceedings of the 61th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Xiao Yu, Yuang Qi, Kejiang Chen, Guoqiang Chen, Xi Yang, Pengyuan Zhu, Weiming Zhang, and Nenghai Yu. 2023. Gpt paternity test: Gpt generated text detection with gpt genetic inheritance. *arXiv preprint arXiv:2305.12519*.

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

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*.

## A Preliminaries for WLLM

In this section, we provide brief preliminaries for Kirchenbauer et al. (2023a). For a given language model $f_{\text{LM}}$ with vocabulary $\mathcal{V}$, the likelihood probability of a token $y_t$ is calculated as follow:

$$\boldsymbol{l}_t = f_{\text{LM}}(\boldsymbol{x}, \boldsymbol{y}_{[:t]}), \tag{2}$$

$$p_{t,i} = \frac{e^{\boldsymbol{l}_t^i}}{\sum_{i=1}^{|\mathcal{V}|} e^{\boldsymbol{l}_t^i}}, \tag{3}$$

where $\boldsymbol{x} = \{x_0, \ldots, x_{M-1}\}$ and $\boldsymbol{y}_{[:t]} = \{y_1, \ldots, y_{t-1}\}$ are a $M$-length tokenized prompt and the generated token sequence, respectively, and $\boldsymbol{l}_t \in \mathbb{R}^{|\mathcal{V}|}$ is the logit vector.

**Watermarking in LM-generated Text.** In the watermarking (Kirchenbauer et al., 2023a), the entire tokens in $\mathcal{V}$ at each time-step are randomly binned into the green $\mathcal{G}_t$ and red groups $\mathcal{R}_t$ in proportions of $\gamma$ and $1-\gamma$ ($\gamma \in (0,1)$), respectively. The method increases the logits of green group tokens by adding a fixed scalar $\delta$, promoting them to be sampled at each position. Thus, watermarked LM-generated text is more likely than $\gamma$ to contain the green group tokens. On the other hand, since humans have no knowledge of the hidden green-red rule, the proportion of green group tokens in human-written text is expected to be close to $\gamma$.

The watermarked text is detected through a one-sided $z$-test by testing the null hypothesis where the text is not watermarked. The $z$-score is calculated using the number of recognized green tokens in the text. Then, the testing text is considered as watermarked if the $z$-score is greater than $z_{\text{threshold}}$. Note that the detection algorithm with the higher $z_{\text{threshold}}$ can result the lower false positive rate (FPR) and reduce Type I errors.

**Spike Entropy** Kirchenbauer et al. (2023a) used *spike entropy* for measuring how spread out a distribution is. Given a token probability vector $p$ and a scalar $m$, spike entropy of $p$ with modulus $m$ is defined as:

$$S(p, m) = \sum \frac{p_k}{1 + m p_k}. \tag{4}$$

## B Watermark Embedding/Detecting Algorithm of SWEET

Algorithms 1 and 2 show the detailed steps of generating a watermark and later detecting it using our selective entropy thresholding method (SWEET).

**Algorithm 1** Generation Algorithm of SWEET

**Input:** tokenized prompt $\boldsymbol{x} = \{x_1, \ldots, x_{M-1}\}$; entropy threshold $\tau \in [0, \log|\mathcal{V}|]$, $\gamma \in (0,1)$, $\delta > 0$;
**for** $t = 0, 1, 2, \ldots$ **do**
  Compute a logit vector $\boldsymbol{l}_t$ by (2);
  Compute a probability vector $\boldsymbol{p}_t$ by (3);
  Compute an entropy $H_t$ by (5);
  **if** $H_t > \tau$ **then**
    Compute a hash of token $y_{t-1}$, and use it as a seed for a random number generator;
    Randomly divide $\mathcal{V}$ into $\mathcal{G}_t$ of size $\gamma|\mathcal{V}|$ and $\mathcal{R}_t$ of size $(1-\gamma)|\mathcal{V}|$;
    Add $\delta$ to the logits of tokens in $\mathcal{G}_t$;
  **end if**
  Sample $y_t$;
**end for**

**Algorithm 2** Detection Algorithm of SWEET

**Input:** tokenized prompt $\boldsymbol{x}$; token sequence to be tested $\boldsymbol{y} = \{y_0, \ldots, y_{N-1}\}$; entropy threshold $\tau \in [0, \log|\mathcal{V}|]$, $\gamma \in (0,1)$, $z_{\text{threshold}} > 0$;
Set $N^h = 0$ and $N_G^h = 0$;
**for** $t = 0, 1, 2, \ldots N-1$ **do**
  Compute a logit vector $\boldsymbol{l}_t$ by (2);
  Compute a probability vector $\boldsymbol{p}_t$ by (3);
  Compute an entropy $H_t$ by (5);
  **if** $H_t > \tau$ **then**
    $N^h \leftarrow N^h + 1$;
    Compute a hash of token $y_{t-1}$, and use it as a seed for a random number generator;
    Recover $\mathcal{G}_t$ and $\mathcal{R}_t$;
    **if** $y_t \in \mathcal{G}_t$ **then**
      $N_G^h \leftarrow N_G^h + 1$;
    **end if**
  **end if**
**end for**
Compute $z$-score by (1);
**if** $z > z_{\text{threshold}}$ **then**
  **return** True; (i.e., $\boldsymbol{y}$ is watermarked)
**else**
  **return** False;
**end if**

Instead of the spike entropy used in WLLM, we use the classical Shannon entropy. Given a token probability distibution vector $p$, the entropy of $p$ is computed by

$$H_t = -\sum p_k \log p_k. \tag{5}$$

## C Proof of Theorem 1

We begin with a lemma from Kirchenbauer et al. (2023a), which predicts the probability of a green list token sampled from a language model employing the watermarking.

In our proof, we predict the lower bounds of $z$-score when detecting watermarks via WLLM or SWEET methods and compare the z-score lower bounds.

**Lemma C.1.** *Suppose $p \in (0,1)^{|\mathcal{V}|}$ is a raw probability vector generated from a language model where $|\mathcal{V}|$ is the vocabulary size. Before sampling $p$, watermarks are embedded by dividing randomly a green list of size $\gamma|\mathcal{V}|$ and a red list of size $(1-\gamma)|\mathcal{V}|$ for some value $\gamma \in (0,1)$. It then promotes the logits of tokens in the green list by $\delta$. When sampling a token index $k$ from this watermarked distribution, the probability that the token is sampled from the green list (considering the randomness of green list) is at least*

$$\mathbb{P}[k \in G] \geq \frac{\gamma e^{\delta}}{1+(e^{\delta}-1)\gamma} S(p, \frac{(1-\gamma)(e^{\delta}-1)}{1+(e^{\delta}-1)\gamma}).$$

Let's begin the proof.

*Proof.* In WLLM, we consider all tokens in $\boldsymbol{y} = \{y_0, \ldots, y_{N-1}\}$ for detection. We can get a lower bound of the number of green list tokens in $\boldsymbol{y}$ by summing the result of Lemma C.1 over the tokens $y_t$. The expectation of the number of green list tokens, $N_g$, in $\boldsymbol{y}$ is at least

$$\mathbb{E}[N_G] \geq \alpha\gamma N\overline{S}. \tag{6}$$

where $\alpha = \frac{e^{\delta}}{1+(e^{\delta}-1)\gamma}$, and $\overline{S} = \sum_{t=1}^{N} S_t/N$.

We can get the lower bound of the $z$-score by applying the $z$-score definition in Eq. 1:

$$z \geq \gamma\sqrt{N}\frac{\alpha\overline{S}-1}{\sqrt{\gamma(1-\gamma)}}. \tag{7}$$

If the entropy threshold is applied, we consider only tokens with entropy values higher than the threshold to be tested. Let $N^h$ be the number of tokens that have higher entropy values. Following Eq. 6 and Eq. 7 again with $N^h$, we can get the lower bound of the $z$-score of SWEET:

$$z \geq \gamma\sqrt{N^h}\frac{\alpha\overline{S^h}-1}{\sqrt{\gamma(1-\gamma)}},$$

where $\overline{S^h} = \sum_{t=1}^{N} S_t \times \mathbb{1}(S_t \geq \tau)/N^h$.

$\overline{S_h} \geq \overline{S}$ is ensured as we ignore all tokens with lower entropy than the threshold. By comparing Eq. 7 and Eq. 8,

$$\gamma\sqrt{N^h}\frac{\alpha\overline{S^h}-1}{\sqrt{\gamma(1-\gamma)}} \geq \gamma\sqrt{N}\frac{\alpha\overline{S}-1}{\sqrt{\gamma(1-\gamma)}},$$

$$\sqrt{\frac{N-N^l}{N}} \geq \frac{\alpha\overline{S}-1}{\alpha\overline{S^h}-1},$$

$$\frac{N^l}{N} \leq 1-(\frac{\alpha\overline{S}-1}{\alpha\overline{S^h}-1})^2,$$

where $N^l = N - N^h$. □

## D Implementation Details

We have used three datasets for our testbeds: HumanEval, MBPP, and DS-1000. They have 164, 500, and 1000 Python code problems, respectively. For our base models, StarCoder and LLaMA2, we use top-$p$ (Holtzman et al., 2020) sampling with $p = 0.95$ for both models, and temperature 0.2 and 0.1, respectively. When generating output for each code problems, we use zero-shot setting in HumanEval and DS-1000 but 3-shot in MBPP. Prompts used in MBPP are similar to the prompt in Austin et al. (2021). For calculating pass@1 scores, we set $n = 40$ for HumanEval and DS-1000, and $n = 20$ for MBPP.

### D.1 DetectGPT

We used two masking models for DetectGPT. When T5-3B is used for DetectGPT, we search hyperparameters for the length of the spans in [1,2,5,10] words, and for the proportion of masks in [5,10,15,20]% of the text. When utilizing SantaCoder, we simulate the single-line fill-in-the-middle task scenario by masking only one line of code per perturbation, which is a task that SantaCoder is trained to perform well. (Fried et al., 2023; Bavarian et al., 2022). We search hyperparameters for the number line to be rephrased in [1,2,3,4]. We make 100 perturbations following the original paper.

### D.2 WLLM and SWEET

Depending on the strength of watermark, trade-off between code functionality and watermarking detectability exists. We search hyperparameters for the ratio of the green list $\gamma$ in [0.1,0.25,0.5], and for the green token promotion value $\delta$ in [0.5,1.0,2.0,3.0,4.0]. For the entropy threshold

| Method | HumanEval | | | |
|---|---|---|---|---|
| | pass@1 | AUROC | TPR | FPR |
| Non-watermarked | **17.3** | - | - | - |
| Non-watermarked (w/ high entropy) | 6.8 | - | - | - |
| EXP-EDIT | **17.1** | 0.612 | 0.110 | <0.05 |
| EXP-EDIT (w/ high entropy) | 7.1 | 0.844 | 0.561 | <0.05 |
| WLLM ($\Delta$PASS@1 $\sim -10\%$)$^\star$ | 15.4 | 0.777 | 0.402 | <0.05 |
| SWEET ($\Delta$PASS@1 $\sim -10\%$)$^\star$ | 15.5 | **0.921** | **0.616** | <0.05 |
| WLLM (AUROC$\geq 0.9$)$^\dagger$ | 9.2 | 0.908 | 0.720 | <0.05 |
| SWEET (AUROC$\geq 0.9$)$^\dagger$ | 15.5 | 0.921 | 0.616 | <0.05 |

Table 2: Results of code generation performance and detection ability in LLaMA2 13B. We calculate pass@1 metrics by generating $n = 40$ examples. Hyperparameters for decoding strategy is top-p decoding with $p = 0.95$ and temperature=0.1, except for baselines with high entropy; temperature=1.0 and top-p=1.0. We set the maximum length of the model generation to 512. This table corresponds to the Table 1 version for LLaMA2, but only for watermark-based methods.

values used in SWEET, we search thresholds in [0.3,0.6,0.9,1.2].

### D.3 EXP-EDIT

In most tasks we have conducted experiments, the length of the generated code hardly exceed 100 tokens. Therefore, considering that length of the watermark key sequence significantly affected the detection speed, we search hyperparameters for the length of the key sequence only in [100, 500]. The block size was set equal to the length of the model output, and the resample size $T = 500$ for all instances. To generate $n$ outputs to calculate pass@k, we shift the watermark key sequence randomly $n$ times. Finally, we set edit distance hyperparameter $\gamma = 0.0$ for EXP-EDIT as used in their paper.

## E Detectability with Varying Code Lengths

We experiment the detection performance across different code lengths. Based on the detectability@T metric proposed in Kirchenbauer et al. (2023b), we evaluate the detection performance within the first $T$ tokens of the machine-generated and human-written code sequences and calculate AUROC scores.

As presented in Figure 5, SWEET demonstrates superior detection performance even in the short code texts. This is particularly important feature in code generation tasks comprised of relatively shorter texts than plain text generation. Moreover, in HumanEval and MBPP, we can observe that the AUROC of SWEET reaches 1.0 with the text length exceeding 70, while none of the baselines could achieve it.

## F Further Pareto Frontier Results on StarCoder/LLaMA2

**HumanEval pass@100.** Figure 8 shows a trade-off between pass@100 score and AUROC at HumanEval task in temperature 0.8. We generated 200 samples in HumanEval to calculate pass@100. The tendency of the Pareto Frontier are the same, SWEET is consistently placed in the front. While pass@100 score is much higher than the pass@1 score at temperature=0.2, we see the range of AUROC remains similar. This indicates temperature does not affect the detection strength of each samples heavily.

**LLaMA2.** Furthermore, Table 2 shows the results on HumanEval when using LLaMA2 13B (a general-purpose LLM), as the backbone for code generation. We can observe similar trends as demonstrated in Figure 9. SWEET in LLaMA2 achieves a higher AUROC than all other baselines while preserving code quality more than WLLM. Consequently, we observe that SWEET also applies to general-purpose LLM, which is not code-specific.

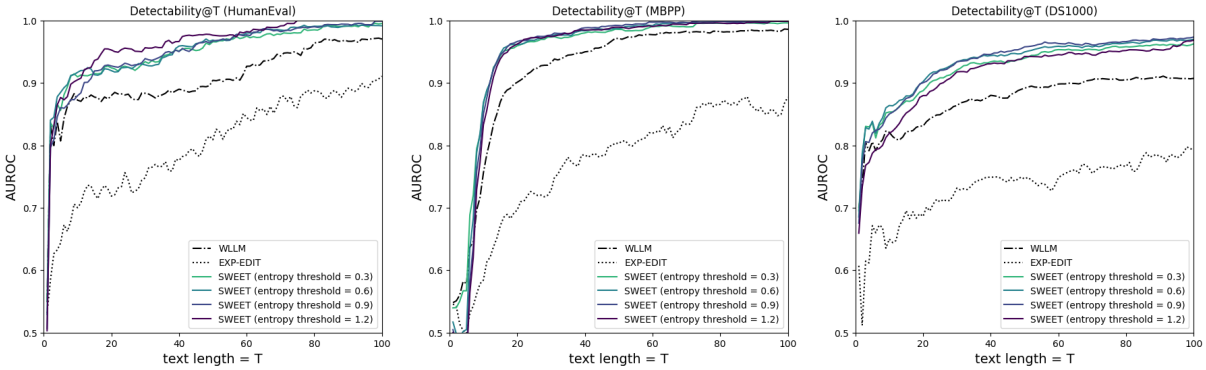


Figure 5: Detectability@T (Kirchenbauer et al., 2023b) at HumanEval, MBPP, and DS-1000. We set $\gamma = 0.25$ and $\delta = 3.0$ for WLLM and SWEET. For EXP-EDIT, we use it with a high entropy setting. When calculating AUROC, we ensure at least 20 code texts of human-written solutions and machine-generated codes, respectively. We can observe that SWEET shows superior detection performance regardless of the text length in all tasks.

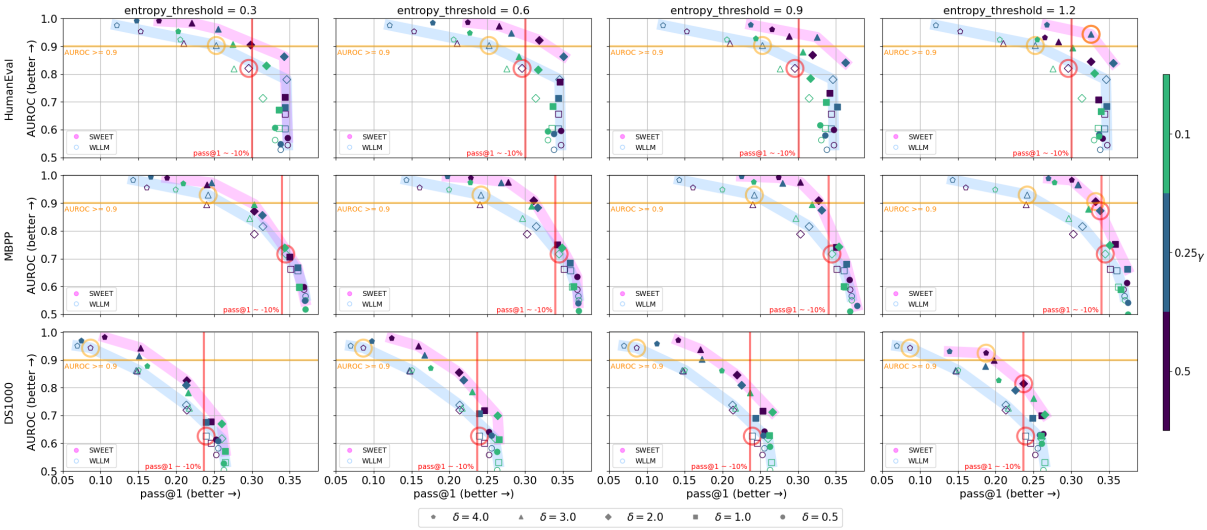


Figure 6: The tradeoff between AUROC and pass@1 of detecting real and generated samples of HumanEval and MBPP datasets. The pink line represents a Pareto frontier of SWEET, while the blue line represents that of WLLM. In all tasks and the entropy threshold configurations, SWEET shows consistent dominance. The red/orange line and circles are the points used in Table 1.

## G More Details about Experiments with General Prompts

All general prompts we mentioned in Sec 6.2 at HumanEval task are listed below: These prompts are chosen randomly without any prompt tuning.

```
def solution(*args):
    """
    Generate a solution
    """
```

```
<filename>solutions/solution_1.py
# Here is the correct implementation of the code exercise
def solution(*args):
```

```
def function(*args, **kargs):
    """
    Generate a code given the condition
    """
```

```
from typing import List

def my_solution(*args, **kargs):
    """
    Generate a solution
    """
```

```
def foo(*args):
    """
    Solution that solves a problem
    """
```

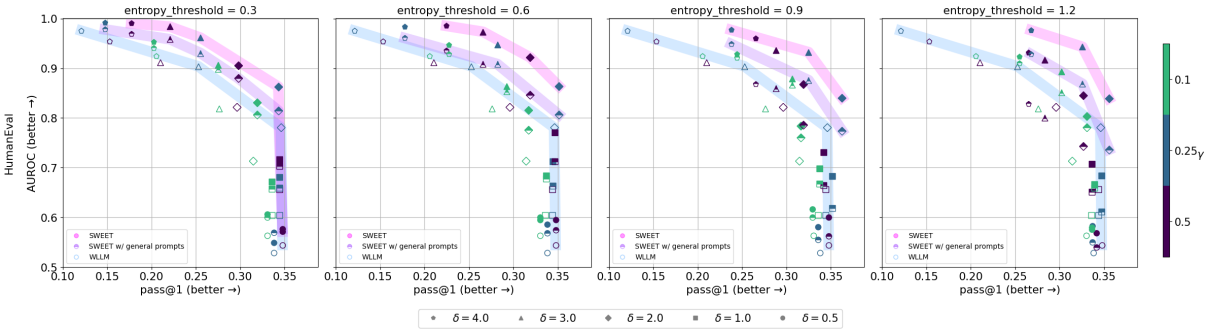


Figure 7: Effect of *general prompts* in SWEET in HumanEval. In this setting, the detector does not know what information would have been included in a prompt if the given sample source code had been model-generated. SWEET appends the sample to the fixed number of 'general prompts' that contain no information except for the format consistent with the answer. The purple line represents the Pareto frontier of the 'General prompts' version SWEET. Our approaches with general prompts still outperform WLLM in both code quality preservation and watermark detection, drawing the Pareto frontiers ahead of those of WLLM.

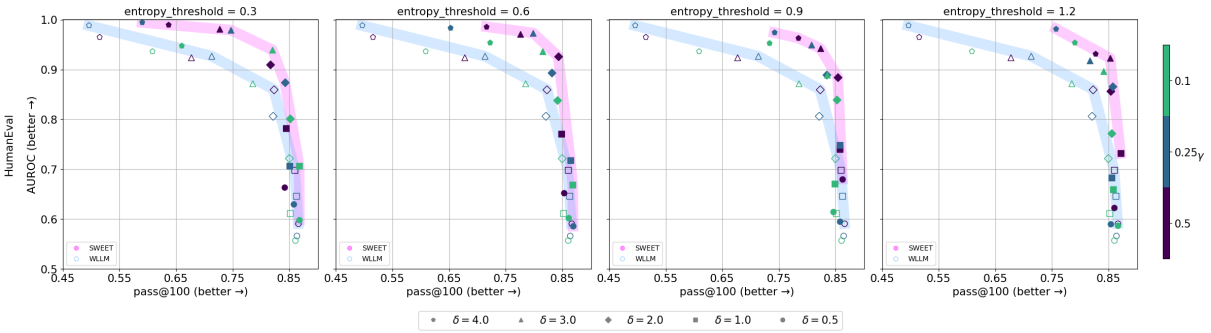


Figure 8: The tradeoff between AUROC and **pass@100** of detecting real and generated samples of HumanEval using temperature of 0.8 instead of 0.2 as other figures. We also generate $n = 200$ outputs for calculating pass@100 scores. The pink line represents a Pareto frontier of SWEET, while the blue line represents a Pareto frontier of WLLM. We observe consistent improvement in SWEET.

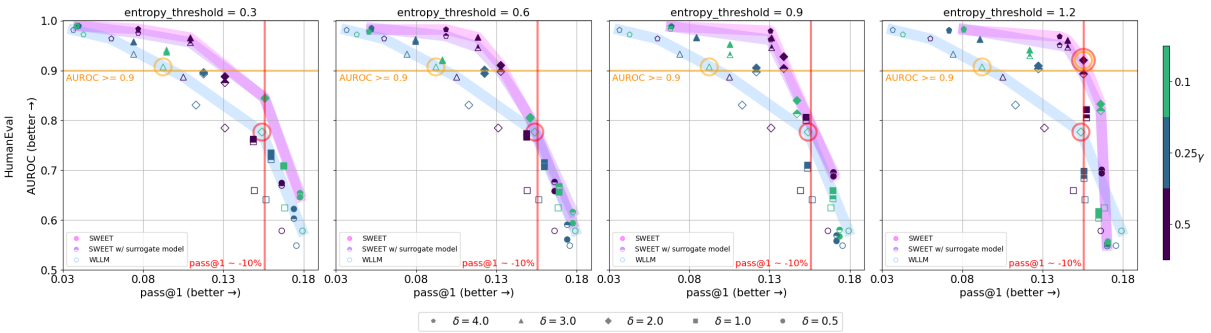


Figure 9: [LLaMa2 13B Results] The tradeoff between AUROC and pass@1 of detecting real and generated samples of HumanEval. The pink line represents a Pareto frontier of SWEET, while the blue line represents a Pareto frontier of WLLM. Additionally, we include the results of the SWEET with the surrogate model (purple line), in which a smaller LM is used to detect watermarks to save computational costs. Our approaches mostly draw Pareto frontiers ahead of those of WLLM, even with the surrogate model. The red/orange line and circles are the points used in Table 2.

## H Analysis of Computation Cost

It is practically important to detect machine-generated text without a huge computational overload. We here analyze computation costs for each baseline and our method.

WLLM does not require any additional com-

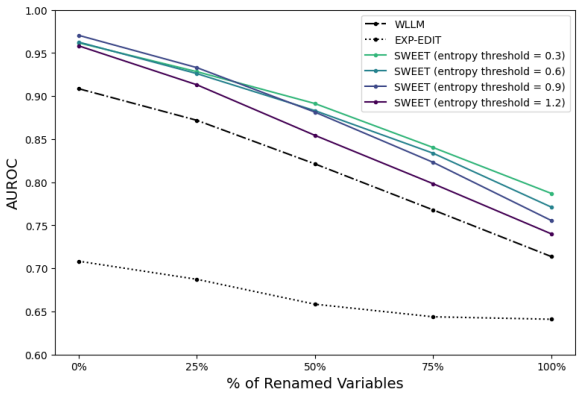


Figure 10: Watermark detection performance on renamed variables in the code. For each watermark method, we choose 273 source codes from the MBPP task, for which three methods succeed in generating with no syntax error. We set $\gamma = 0.25$ and $\delta = 3.0$ for WLLM and SWEET. For EXP-EDIT, we search the hyperparameter for the block size in [20,30,40] with a high entropy setting. We use five random seeds for renaming and calculate the average AUROC scores.

putation as it only needs a random number generator and a seed number to put. On the other hand, all zero-shot post-hoc detection methods excluding DetectGPT need at least one forward pass of that LLM. DetectGPT needs to run forward passes as much as the number of perturbations for increased accuracy (the original paper generated 100 perturbed samples, so we did the same). Our method needs one time forward pass to calculate the entropy, which is the same with zero-shot post-hoc detection methods except for DetectGPT. However, we demonstrated that our method outperforms baselines even when utilizing a smaller surrogate model (Sec 6.3), indicating the capability of computationally more efficient employment. On the other hand, while EXP-EDIT does not need LLM for detecting watermarks, it requires measuring the Levenshtein distance to compute the test statistic, which demands an extensive calculation of $O(mnk^2)$, where $m$ be the length of the target text, $n$ be the length of the watermark key sequence, and $k$ be the block size. Moreover, $T = 500$ times of test statistic is also necessary for reporting the p-value. Although these computations do not require LLM and can be implemented in parallel, one can consider the computation cost of EXP-EDIT as high.

# I Analysis of Lexical Type Distributions

Watermarking a text without degrading its quality is possible when many candidates are alternatively available. In code generation, it is challenging to achieve this, so SWEET selectively apply watermarking only on high entropy, i.e., when there are many candidates. Using Python built-in tokenize module[10], we here tokenize outputs of our SWEET method and analyze the distributions of lexical types both above and below the entropy threshold.

## I.1 List of Lexical Types

Below is the list of lexical types we use for analysis and corresponding examples. All list of types the tokenize module actually emits can be found in https://docs.python.org/3/library/token.html. We merged and split the original types.

- NAME : identifier names, function names, etc.
- OP : operators, such as {, [ ( +, =, etc.
- INDENT : we merge NEWLINE, DEDENT, INDENT, NEWLINE, and NL.
- RESERVED : split from NAME. In Python docs, they are officially named *keywords*.
- BUILT-IN : split from NAME. Please refer to Python docs[11].
- NUMBER
- STRING
- COMMENT
- FUNCNAME : split from NAME. We manually build a list of function name almost being used only for function. For examples, append(), join(), split() functions are included.

## I.2 Lexical Types Distributions Above Threshold

Figure 11 shows lexical types distributions of output tokens above the entropy threshold (i.e., watermarked tokens) across seven thresholds. As the entropy threshold rises, the proportion of NAME type tokens increases by the most (26%p to 63%p). Intuitively, this can be easily understood, considering there would be many alternative candidates

[10] https://docs.python.org/3/library/tokenize.html

[11] https://docs.python.org/3/library/functions.html#built-in-functions

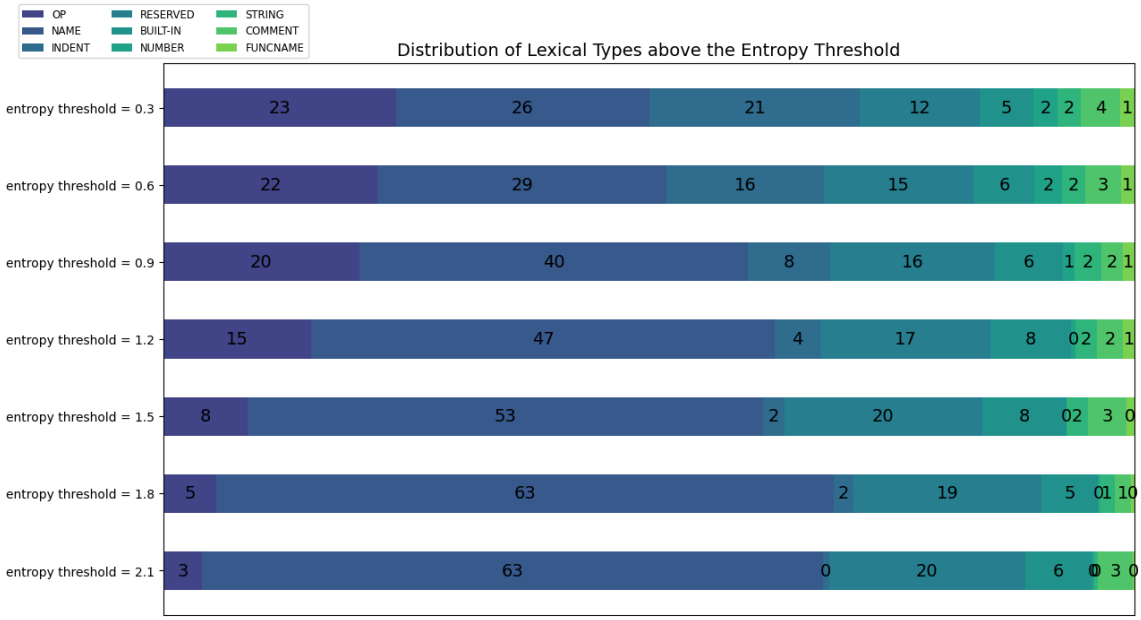


Figure 11: Distribution of lexical types of SWEET output on HumanEval task. We draw examples when $\gamma = 0.25$ and $\delta = 3.0$. The proportion of NAME type tokens increases the most while that of INDENT type tokens converges to zero.

for defining identifier names. Unfortunately, this would lead to vulnerability to an adversarial attack on watermarking, such as changing variable names. Following the NAME type, the ratio of the RESERVED type also increases slightly (12%p to 20%p), meaning that the model has multiple choices of logical flow in code generation, considering RESERVED tokens usually decide code execution flow.

### I.3 Lexical Types Distributions Below Threshold

Figure 12 shows lexical types distributions of output tokens below the entropy threshold. In contrast to the distributions above the threshold, NAME and RESERVED types do not increase as the threshold rises. Meanwhile, the proportion of INDENT types slightly increases (18%p to 22%p), indicating that the model has more confidence in the rules, such as indentation.

## J Further Analysis of Breakdown of Post-hoc methods

The performance of post-hoc detection methods in the machine-generated code detection task is surprisingly low compared to their performance in the plain text domain. In both HumanEval and MBPP, none of the post-hoc baselines have an AUROC score exceeding 0.6, and the TPR is around 10% or even lower. In this section, we analyze the failures of post-hoc detection baselines.

**Out-Of-Domain for classifiers**. Methods leveraging trained classifiers, such as GPTZero and OpenAI Classifier, inherently suffer from out-of-domain (OOD) issues (Guo et al., 2023; Yang et al., 2023). Since the machine-generated code detection problems are relatively under explored, we can conjecture that there are not enough examples of machine-generated code for training, especially even though we do not know of the dataset on which GPTZero was trained.

**Relatively Short Length of Code Blocks**. DetectGPT presumes the length of the text being detected as near paragraph length. OpenAI Classifier released in 2023 (OpenAI, 2023b) takes only text longer than 1,000 tokens. Even in the WLLM and their following paper (Kirchenbauer et al., 2023b), the length is one of the prime factors in detection and is used in a metric, detectability@T. Despite the importance of the length, in our experiments, the length of the generated code text is generally short. The token lengths generated by the model were are 59 and 49 tokens on average for HumanEval and MBPP, respectively. Unless embedding some signals in the text intentionally, like WLLM and ours, it seems that it is challenging for post-hoc methods to detect short text.

**Failures in DetectGPT**. Specifically, in Detect-

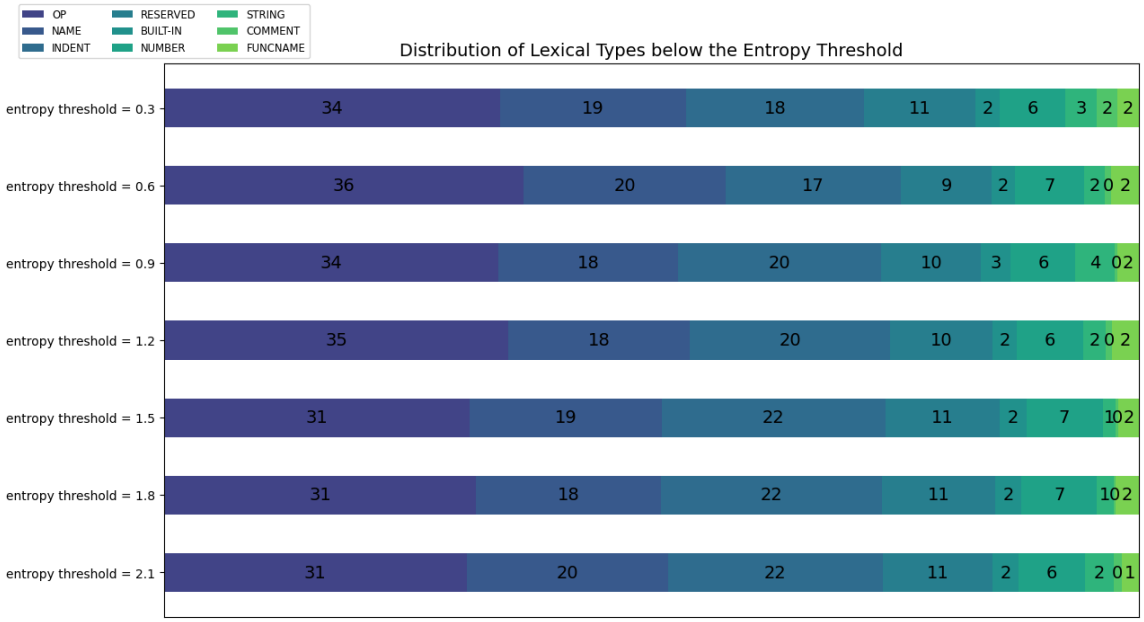


Figure 12: Distribution of lexical types of SWEET output on HumanEval task. We draw examples when $\gamma = 0.25$ and $\delta = 3.0$. In contrast to the distributions above the threshold, there is almost no distribution change.

GPT, we attribute the failure to detect machine-generated code to poor estimation of perturbation curvature. We hypothesize two reasons for this. Firstly, considering the nature of the code, it is challenging to rephrase a code while preserving its meaning or functionality. To minimize the degradation of perturbation, we use SantaCoder for the masking model and paraphrase only one line of code at a time. Yet, in most cases, the rephrased code is either identical to its original or broken in functionality. Secondly, LLMs have not achieved as satisfactory code generation performance as plain text generation. Hence, the base and masking models cannot draw meaningful curvature.