SEAL: STATISTICAL EMBEDDING ALIGNMENT FOR CODE WATERMARKING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models can now generate high-quality code at scale, making reliable source tracing increasingly important. Code watermarking, however, is more restrictive than text watermarking: generated code must remain executable, functionally correct, and robust to common edits. In contrast to the existing code watermarking methods that rely on embedding the signal in token choices, syntax-aware rules, or decoding-time biases, tying the watermark to surface form, we propose **SEAL**, a semantic-space watermarking method for code generation that embeds the signal through candidate selection in code embedding space. Rather than retraining model parameters, decoding logits, or alternating the final program, SEAL selects among model-generated candidates whose semantic projection signs align with a secret key. We further establish **two theoretical results** proving that SEAL produces a statistically detectable signal that strengthens under aggregation across generated functions, while remaining robust to code transformations that preserve most semantic watermark signs. Experiments on HumanEval and MBPP with CodeLlama-7B and StarCoder2-7B show that SEAL is competitive with SWEET and STONE. Aggregation improves low-FPR detection, and SEAL remains detectable under formatting changes, variable renaming, and code rewrites. Our results show that semantic embedding-space selection offers a practical path toward robust code watermarking, where each generated function serves as the watermark-carrying unit and the signal is encoded in the semantic representation.

1 INTRODUCTION

The recent development of large language models (LLMs) has significantly improved their ability to generate high-quality code, powering applications in software development, program synthesis, and automated assistance Zhao et al. (2023a); Tang et al. (2023); Jiang et al. (2026); Dolcetti et al. (2026); Kharma et al. (2026). As these models become more widely used in real-world scenarios, distinguishing machine-generated code from human-written code becomes increasingly difficult, raising concerns about provenance, attribution, and misuse Yang et al. (2024); Li et al. (2023). This creates a need for watermarking methods that can reliably identify the source of generated code Cao et al. (2025); Li et al. (2024).

Watermarking methods refer to the process of embedding identifiable signals into generated outputs that are not easily detectable but can be identified and verified algorithmically Zhao et al. (2025); Gu et al. (2025); Xu et al. (2024); Suresh et al. (2024). In natural language generation, a common approach is to bias token selection during decoding Kirchenbauer et al. (2023); Zhao et al. (2023b); Rosenblum et al. (2011); Lu et al. (2024). However, applying these methods directly to code is not as straightforward. Programming languages impose strict syntactic and semantic constraints, and even small changes to token selection can break compilation or alter program behavior Lee et al. (2024); Hoang et al. (2024). This makes it difficult to embed watermark signals without affecting functional correctness. Several approaches have been proposed to address this challenge. CodeMark Sun et al. (2023) embeds watermark signals during training by introducing dependencies between semantically equivalent code patterns, allowing the model to learn these correlations. SWEET Lee et al. (2024) instead applies watermarking during generation, restricting modifications to token positions with higher uncertainty. STONE Kim et al. (2026) further refines this approach by avoiding changes to syntax-critical tokens, improving functional reliability. However, as pointed out and assessed by a Systematization of Knowledge (SoK) paper this year Cao et al. (2025), current code watermarking

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

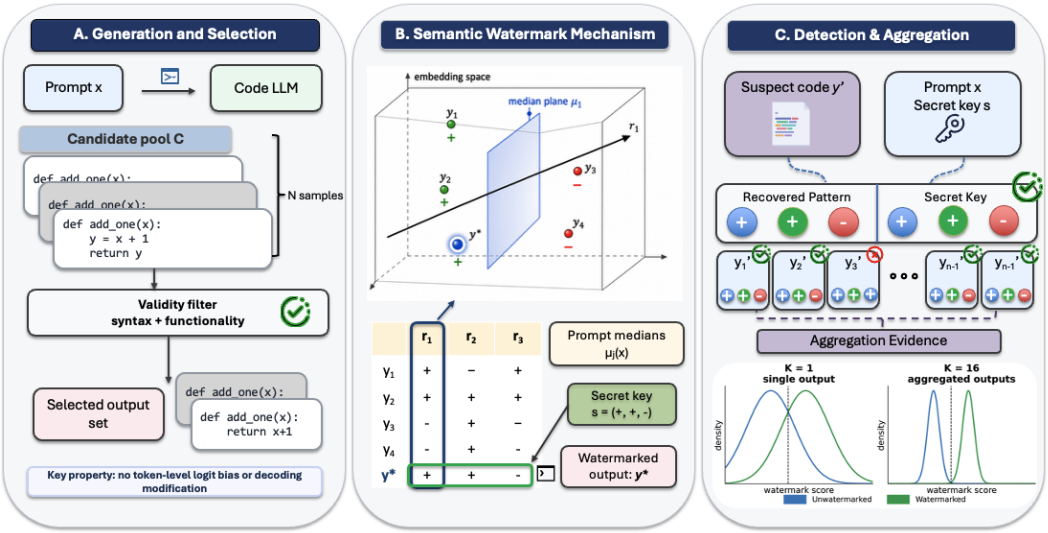

Figure 1: Overview of SEAL. (A) For each prompt x, the code model samples a candidate pool $\mathcal{C}(x) = \{y_i\}_{i=1}^{N}$ and filters candidates by syntax and functionality. (B) Each valid candidate y_i is embedded into a semantic code space and projected onto secret directions. Prompt-specific medians split each channel into positive and negative signs, and SEAL selects the valid candidate $y^\star$ whose sign pattern best matches the secret key $\mathbf{s}$. (C) During detection, the verifier uses the prompt x and secret key $\mathbf{s}$ to recover the watermark pattern from a suspect program y'. Evidence is aggregated across generated functions, reducing detector uncertainty and improving separation between watermarked and unwatermarked code.

signals can "easily be erased by minor semantic-preserving edits" and "future work should explore hybrid watermarking schemes that combine statistical bias with semantic-level patterns". It points to **two central challenges** for code watermarking: the watermark signal must survive semantics-preserving edits, and it must do so without imposing excessive distortion on the generated code on the token-level. Existing methods largely remain tied to shifting the model output's statistical distribution, either through training data biasing or token-level decoding Lee et al. (2024); Kim et al. (2026). This creates a trade-off between maintaining correctness and embedding a strong, detectable signal. It also makes the watermark sensitive to common code edits, since local token choices can change while the program behavior remains the same Cao et al. (2025).

In this paper, we tackle these two challenges by providing, to the best of our knowledge, the first semantic-space watermarking technique for generated code and a theoretical robustness performance guarantee. Rather than modifying the model or its output distribution, we treat the model as fixed and embed watermark signals through a carefully designed statistical selection process among semantically equivalent candidates. The key observation, inspired by recent developments in semantic-space watermarking for natural text Huo et al. (2025); Giboulot and Furon (2024); Yu et al. (2025), is that a single coding prompt and task can admit multiple valid implementations. By sampling multiple candidates and selecting them based on our pre-selected secret signal, SEAL can embed a watermark without altering the generation process, which can later be detected through statistical analysis of the selected output. Our **main contributions** can be summarized as follows:

1. **First semantic-space code watermarking:** To the best of our knowledge, SEAL is the first method to watermark LLM-generated code through semantic embedding-space selection rather than token-level or syntax-level biasing.
2. **No output distributional distortion or retraining:** SEAL embeds the watermark by choosing among valid model-generated candidates, rather than changing token probabilities, retraining the model, or rewriting the output. This makes the method compatible with black-box generation pipelines while preserving the model's original decoding behavior.
3. **Theoretical analysis on detection and robustness:** SEAL leaves a small secret-key-dependent trace in each selected function. We proved that this signal grows through aggre-

gation across generated functions and are proven to remain positive under transformations that preserve more than half of the recovered semantic signs.

4. **Multi-function attribution suited for realistic settings:** SEAL treats each function as a watermark-carrying unit, but its verification naturally strengthens when applied to files, modules, or projects containing multiple generated functions. This matches practical code generation settings, where attribution is often performed over a body of generated code rather than a single isolated completion.

2 Related Work

Due to the strict syntactic and semantic constraints of programming languages and functional correctness, code watermarking has diverged from natural language watermarking Guo et al. (2025); Sun et al. (2023); Suresh et al. (2024). Several early approaches have been proposed as token-level watermarking methods. The earliest work, CodeMark Sun et al. (2023), was introduced in 2023, which embeds a watermark during training by introducing dependencies between semantically equivalent code patterns Sun et al. (2023). CodeMark first identifies pairs of equivalent patterns (e.g., alternative expressions of the same operation) in a block of code, then it modifies the training corpus so that one pattern is more likely to appear when the other is present. As a result, a model trained on this modified skewed data learns the correlation. During watermark detection, the model is queried with inputs containing a single pattern, and the occurrence of the paired pattern is measured using statistical tests. While this approach distributes the watermark signal across program structure and improves robustness compared to local token-level signals Cao et al. (2025), it fundamentally relies on altering the underlying data distribution during training. This requires access to and modification of the training process, and the watermark exists only implicitly in the model's learned behavior, rather than being directly encoded in individual generated samples.

SWEET Lee et al. (2024) later introduced a token-level watermarking method for code generation. Specifically, it observes that, compared to natural language, code generation tends to be extremely low entropy, meaning that many token positions allow only a small number of valid choices Gu et al. (2025); Lu et al. (2024). Therefore, applying watermarks uniformly would either break the correctness of the code or result in a weak watermark signal Lee et al. (2024). SWEET addresses this by identifying and applying watermarks only to relatively high-entropy tokens, while skipping low-entropy positions. STONE Kim et al. (2026) was proposed to further improve code watermarking by incorporating syntax awareness. Specifically, it identifies tokens that affect program structure, such as keywords and control-flow elements, and avoids modifying them. The watermark is instead applied only to tokens that are less likely to affect code execution, which reduces the chance of generating invalid or incorrect code compared to earlier token-level methods. Although SWEET and STONE improve the trade-off between correctness and detectability, both methods still rely on modifying the token-level distribution during decoding, and the watermark signal remains tied to local token choices rather than higher-level program structure.

3 Method

3.1 Problem Formulation

Let x denote a programming prompt, such as a function signature with a natural language specification, and let $p_\theta(y \mid x)$ be a pretrained code generation model. Our goal is to produce a function body y that satisfies the programming task while carrying a secret message

$$\mathbf{s} = (s_1, s_2, \ldots, s_m), \qquad s_j \in \{-1, +1\}, \tag{1}$$

where m is the number of semantic watermark channels assigned to each generated function.

SEAL formulates watermarking as a constrained selection problem. Rather than modifying token logits, changing code, or retraining the model, SEAL samples multiple candidate programs from the original model and selects a valid candidate whose semantic embedding matches the pre-defined secret message. The watermark is therefore embedded through selection among model generated candidates, not through direct modification of the generated code.

3.2 WATERMARK EMBEDDING

For each prompt x, SEAL samples a pool of N candidate completions:

$$\mathcal{C}(x) = \{y_1, y_2, \ldots, y_N\}, \qquad y_i \sim p_\theta(\cdot \mid x). \tag{2}$$

In our implementation, each candidate is treated as a function body. This removes superficial variation from duplicated signatures, prompt echoes, markdown fences, and surrounding file context, and focuses watermarking on the executable implementation.

Each sampled candidate is passed through a validity filter. We first clean formatting artifacts and normalize indentation. We then require syntactic validity by parsing the reconstructed program. When benchmark tests are available, we additionally require functional validity by running the associated tests. Let the surviving pool be $\tilde{\mathcal{C}}(x) \subseteq \mathcal{C}(x)$ and SEAL only selects from this filtered pool. The same filtering rule is used during verification. In benchmark settings, this means that the verifier has access to the same unit tests; in deployment settings without tests, both embedding and verification use syntax-based filtering only. This keeps the prompt median computation consistent between watermarking and verification.

SEAL then embeds each surviving candidate into a semantic code representation space. Let $f_{\text{enc}} : \mathcal{Y} \to \mathbb{R}^d$ be a pretrained code encoder. For each candidate $y_i \in \tilde{\mathcal{C}}(x)$, we compute

$$\mathbf{z}_i = f_{\text{enc}}(y_i). \tag{3}$$

This design moves the watermark signal away from surface token patterns and into a representation space that better reflects program structure and behavior. In our experiments, we use CodeBERT Feng et al. (2020) as the encoder because it provides pretrained representations of source code that capture semantic and structural information beyond surface token patterns, making it a natural space for function-level watermarking for correctness preservation.

The secret key determines a set of projection directions in embedding space:

$$\mathcal{K} = \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_m\}, \qquad \mathbf{r}_j \in \mathbb{R}^d. \tag{4}$$

Each direction $\mathbf{r}_j$ defines one semantic watermark channel. For every candidate y_i and channel j, SEAL computes a projection score $g_j(y_i) = \langle \mathbf{z}_i, \mathbf{r}_j \rangle$. For each prompt x, SEAL then computes the median projection score over the surviving candidate pool:

$$\mu_j(x) = \text{median}_{y_i \in \tilde{\mathcal{C}}(x)} g_j(y_i). \tag{5}$$

The prompt-specific median defines a balanced decision boundary for channel j. Each candidate receives a sign according to which side of the median it falls on, with ties assigned to $+1$:

$$q_j(y_i; x) = \begin{cases} +1, & g_j(y_i) \geq \mu_j(x), \\ -1, & g_j(y_i) < \mu_j(x). \end{cases} \tag{6}$$

Thus, every candidate receives a semantic sign pattern

$$\mathbf{q}(y_i; x) = (q_1(y_i; x), \ldots, q_m(y_i; x)). \tag{7}$$

The median construction is central to our method. For each prompt and channel, it splits the candidate pool into approximately two balanced groups. As a result, an unwatermarked candidate should match a fixed secret sign with probability close to one half. This provides SEAL with a simple statistical baseline for detection and enables principled theoretical analysis.

We then select a candidate whose sign pattern agrees with the secret message $\mathbf{s}$. In the ideal case, a candidate satisfies all m semantic constraints:

$$q_j(y_i; x) = s_j, \qquad j = 1, \ldots, m. \tag{8}$$

And the selected output lies on the secret side of each prompt median plane. However, because the candidate pool is finite, an exact match may not always exist.

Our implementation therefore uses sequential median filtering. Starting from $\mathcal{P}_0 = \tilde{\mathcal{C}}(x)$, for each channel $j = 1, \ldots, m$, we keep the candidates that match the secret sign whenever this set is non-empty:

$$\mathcal{P}_j = \begin{cases} \{y_i \in \mathcal{P}_{j-1} : q_j(y_i; x) = s_j\}, & \text{if non-empty,} \\ \mathcal{P}_{j-1}, & \text{otherwise.} \end{cases} \tag{9}$$

The final output is selected from $\mathcal{P}_m$. This greedy procedure prioritizes agreement with the secret signs in the fixed channel order specified by the key. When multiple candidates remain, the one with the highest agreement score is selected:

$$\sum_{j=1}^{m} \mathbb{I}\left[q_j(y_i; x) = s_j\right]. \tag{10}$$

This reconciles the formal selection rule with the implemented sequential median filtering procedure.

This selection rule optimizes the number of keyed semantic agreements, not the geometric distance to each median boundary. This choice matches our detection statistic and robustness analysis, which are both based on sign agreement. A margin-aware variant could further prefer candidates farther from the median planes, but it may also select less typical candidates from the generation pool. We therefore use agreement-based selection as the main method and treat margin-aware selection as a possible robustness extension.

3.3 Watermark Verification

SEAL uses prompt conditioned verification. Given a suspect program $y^\star$, the verifier tests whether it carries the secret message associated with prompt x and key $\mathcal{K}$. The verifier reconstructs the same semantic channels by using the same encoder, key, sampling configuration, random seed, and candidate filtering rule used during watermark embedding.

In the primary verification setting, the verifier deterministically reconstructs the filtered reference pool $\tilde{\mathcal{C}}_{\mathrm{ver}}(x) = \tilde{\mathcal{C}}(x)$ and therefore obtains the same prompt specific medians $\mu_j(x)$ used during embedding. If the logged seed is unavailable, the verifier may instead resample a new finite pool from the same prompt conditioned generation distribution. This can perturb individual median estimates and flip some recovered signs, but the aggregate statistic is expected to remain stable as the pool size, number of semantic channels, or number of verified programs grows, since these fluctuations are averaged across the final detection score. The suspect program is embedded as $\mathbf{z}^\star = f_{\mathrm{enc}}(y^\star)$. For each channel, the verifier recovers a sign:

$$\hat{s}_j(y^\star; x) = \mathrm{sign}\left(\langle \mathbf{z}^\star, \mathbf{r}_j \rangle - \mu_j(x)\right). \tag{11}$$

The recovered pattern is

$$\hat{\mathbf{s}}(y^\star; x) = (\hat{s}_1, \ldots, \hat{s}_m). \tag{12}$$

The single output agreement score is

$$A(y^\star; x) = \frac{1}{m} \sum_{j=1}^{m} \mathbb{I}\left[\hat{s}_j(y^\star; x) = s_j\right]. \tag{13}$$

A score near 1 indicates strong agreement with the secret message, while an unwatermarked output should concentrate near $1/2$ under the balanced null model.

For a collection of K suspect programs associated with prompts $x_1, \ldots, x_K$, SEAL aggregates evidence across outputs:

$$S = \sum_{k=1}^{K} \sum_{j=1}^{m} \mathbb{I}\left[\hat{s}_j(y_k; x_k) = s_{k,j}\right]. \tag{14}$$

The corresponding aggregate agreement score is $A_K = \frac{S}{Km}$. This setting matches practical code generation scenarios where a user or system produces multiple functions, files, or completions from the same model in one setting. A single generated function may contain only weak evidence, but evidence accumulates across multiple generated units. We prove this property of SEAL below based on our statistical detection guarantee.

3.4 Threat Model

We consider a private-key verification setting broadly consistent with prior watermarking work Lee et al. (2024); Kim et al. (2026); Huo et al. (2025); Jalil and Mirza (2009). Given a suspect program, an authorized verifier tests whether it was produced under a secret watermarking key. SEAL

further uses prompt-conditioned verification: the verifier has access to the associated prompt, the secret key, the code encoder, the sampling configuration, and the same candidate filtering rule used during watermarking. When benchmark unit tests are available, both watermarking and verification use the same functionality-based filter; otherwise, both rely only on syntax-based filtering. This setting matches code generation services and benchmark auditing, where prompts, outputs, decoding parameters, random seeds, filtering rules, and available tests can be logged Jain et al. (2024); Sander et al. (2025).

We assume that the adversary can observe and modify the generated program, but does not know the secret key or the selected semantic channels. The adversary aims to remove or weaken the watermark while keeping the generated code useful. We therefore focus on functionality-preserving transformations, including formatting changes, variable renaming, comment edits, dead-code insertion, and semantics-preserving rewrites. For verification, our primary setting assumes that the verifier can reconstruct the candidate pool from the logged prompt, random seed, sampling configuration, and filtering rule, or equivalently, store the prompt-specific medians as verification metadata. If exact reconstruction is unavailable, re-sampling a new pool may slightly shift individual median estimates and flip some recovered signs. On the larger scale, the median should not alter much. We nonetheless use deterministic reconstruction as the main setting and do not claim prompt-free verification.

4 Theoretical Analysis

4.1 Statistical Detection and Aggregation

SEAL detection can be viewed as a private-key hypothesis test over recovered semantic signs. The prompt-specific median defines a binary decision boundary for each channel, so an unwatermarked output is expected to agree with the secret sign no better than random guessing. In contrast, SEAL selection introduces a small key-dependent agreement bias. Theorem 1 shows that this bias becomes easier to detect when evidence is aggregated across generated functions.

Theorem 1. SEAL detection improves with aggregation. Let K be the number of generated functions used for verification and let m be the number of semantic channels per function. Let

$$\hat{s}_j(y_k; x_k) = \mathrm{sign}(\langle f_{\mathrm{enc}}(y_k), \mathbf{r}_j \rangle - \mu_j(x_k))$$

denote the verifier's recovered sign for output y_k on channel j. Define

$$X_{k,j} = \mathbb{I}[\hat{s}_j(y_k; x_k) = s_j] \,.$$

Suppose that, under H_0, $\mathbb{E}[X_{k,j}] = 1/2$ and the aggregate covariance among the Km match indicators is asymptotically negligible, i.e.,

$$\sum_{(k,j)\neq(k',j')} \mathrm{Cov}_{H_0}(X_{k,j}, X_{k',j'}) = o(Km).$$

Suppose that, under H_1, $\mathbb{E}[X_{k,j}] = 1/2 + \delta$ for some $\delta > 0$. Define

$$S_K = \sum_{k=1}^{K} \sum_{j=1}^{m} X_{k,j}, \qquad Z_K = \frac{S_K - Km/2}{\sqrt{Km/4}}.$$

Then

$$\mathbb{E}_{H_0}[Z_K] = 0, \qquad \mathrm{Var}_{H_0}(Z_K) = 1 + o(1), \qquad \mathbb{E}_{H_1}[Z_K] = 2\delta\sqrt{Km}.$$

Therefore, for fixed m and $\delta > 0$, the expected detection strength increases with K at a rate of $\sqrt{K}$. We provide the full proof in Appendix A.1.

The theorem formalizes the aggregation effect observed in our experiments in Table 1 and Table 2. A single generated function may contain only a weak semantic watermark signal, but aggregating evidence across multiple functions reduces uncertainty in the detector statistic.

4.2 Robustness

We next analyze how SEAL behaves under code transformations, where our analysis models an edit as a perturbation in the recovered semantic sign pattern. This gives SEAL a robustness guarantee: as long as the transformation does not randomize more than half of the watermark signs, the expected detection signal remains positive.

Table 1: Controlled detectability comparison on CodeLlama-7B-Instruct. We report AUROC, TPR@1%FPR, and TPR@5%FPR using watermarked outputs as positives and unwatermarked model outputs as negatives. For all methods, K denotes aggregation over K generated outputs before computing detection metrics.

HumanEval					MBPP				
Method	K	AUROC	TPR@1%	TPR@5%	Method	K	AUROC	TPR@1%	TPR@5%
STONE	1	0.534 ± 0.017	0.016 ± 0.004	0.070 ± 0.010	STONE	1	0.577 ± 0.014	0.020 ± 0.005	0.101 ± 0.008
STONE	4	0.548 ± 0.020	0.016 ± 0.016	0.070 ± 0.008	STONE	4	0.610 ± 0.021	0.023 ± 0.008	0.118 ± 0.027
STONE	8	0.559 ± 0.024	0.026 ± 0.011	0.075 ± 0.011	STONE	8	0.637 ± 0.031	0.027 ± 0.008	0.126 ± 0.053
STONE	16	0.576 ± 0.027	0.052 ± 0.012	0.085 ± 0.012	STONE	16	0.682 ± 0.035	0.059 ± 0.021	0.196 ± 0.040
SWEET	1	0.621 ± 0.007	0.047 ± 0.003	0.136 ± 0.006	SWEET	1	0.754 ± 0.006	0.169 ± 0.019	0.321 ± 0.012
SWEET	4	0.663 ± 0.009	0.049 ± 0.009	0.153 ± 0.025	SWEET	4	0.848 ± 0.004	0.298 ± 0.020	0.491 ± 0.036
SWEET	8	0.696 ± 0.006	0.066 ± 0.006	0.212 ± 0.040	SWEET	8	0.905 ± 0.006	0.431 ± 0.023	0.645 ± 0.027
SWEET	16	0.750 ± 0.012	0.098 ± 0.071	0.216 ± 0.052	SWEET	16	0.968 ± 0.002	0.618 ± 0.045	0.822 ± 0.042
SEAL	1	0.739 ± 0.023	0.028 ± 0.009	0.146 ± 0.048	SEAL	1	0.758 ± 0.028	0.037 ± 0.026	0.190 ± 0.052
SEAL	4	0.896 ± 0.023	0.285 ± 0.061	0.472 ± 0.037	SEAL	4	0.907 ± 0.020	0.275 ± 0.071	0.575 ± 0.071
SEAL	8	0.964 ± 0.013	0.500 ± 0.132	0.767 ± 0.076	SEAL	8	0.980 ± 0.009	0.600 ± 0.141	0.850 ± 0.071
SEAL	16	$\mathbf{0.993} \pm 0.006$	$\mathbf{0.767} \pm 0.153$	$\mathbf{0.933} \pm 0.058$	SEAL	16	$\mathbf{0.999} \pm 0.001$	$\mathbf{1.000} \pm 0.001$	$\mathbf{1.000} \pm 0.001$

Theorem 2. SEAL remains detectable as long as the attacker flips fewer than 50% of the semantic watermark signs. Let a watermarked program have per-channel agreement probability

$$\Pr[\hat{s}_j = s_j] = \frac{1}{2} + \delta, \qquad \delta > 0.$$

Let F_j denote the event that the attack flips the recovered semantic sign in channel j, so that

$$\epsilon = \Pr[F_j] = \Pr[\tilde{s}_j \neq \hat{s}_j].$$

Assume that F_j is independent of the original agreement event $\{\hat{s}_j = s_j\}$. Then the post-attack agreement probability is

$$\Pr[\tilde{s}_j = s_j] = \frac{1}{2} + \delta(1 - 2\epsilon).$$

Consequently, for K generated functions and m semantic channels, the expected post-attack detection statistic satisfies

$$\mathbb{E}[Z_{\text{attack}}] = 2\delta(1 - 2\epsilon)\sqrt{Km}.$$

Thus, the watermark signal remains positive whenever $\epsilon < 1/2$. We provide the full mathematical proof and a geometric interpretation in the Appendix A.1.

The theorem shows that SEAL is robust to transformations that preserve more than half of the recovered semantic signs. The detection signal is not binary: it decays linearly with the bit-flip rate through the factor $(1 - 2\epsilon)$, and aggregation over K generated functions can compensate for moderate degradation. This matches the intended behavior of semantic watermarking, where small representation-preserving edits weaken the signal but do not immediately destroy it. We demonstrate our superior robustness results in Table 5.

5 Experiments

5.1 Experimental Setup

Datasets and Models. We evaluate SEAL on HumanEval Chen et al. (2021) and MBPP Austin et al. (2021). HumanEval contains 164 Python function generation tasks, while MBPP contains 974 Python programming tasks with natural language descriptions and unit tests. We further report selected DS-1000 Lai et al. (2023) results in Appendix A.2. We evaluate CodeLlama-7B-Instruct Roziere et al. (2023) and StarCoder2-7B Lozhkov et al. (2024) as base code generation models.

SEAL Parameter Settings. All embedding runs use the same core configuration unless noted otherwise on Ubuntu 22.04.5 LTS (Jammy Jellyfish) with AMD EPYC 7763 64 Cores with 1 48 GB Nvidia A6000 GPU. For each prompt, SEAL embeds an m-bit semantic watermark with $m = 8$ semantic channels. Each surviving candidate is encoded using CodeBERT Feng et al. (2020), with mean pooling over the last hidden states to obtain a single vector representation. For each prompt,

Table 2: Controlled detectability comparison on StarCoder2-7B. We report AUROC, TPR@1%FPR, and TPR@5%FPR using watermarked outputs as positives and unwatermarked model outputs as negatives. For all methods, K denotes aggregation over K generated outputs before computing detection metrics.

HumanEval					**MBPP**				
Method	K	AUROC	TPR@1%	TPR@5%	Method	K	AUROC	TPR@1%	TPR@5%
STONE	1	0.576	0.032	0.077	STONE	1	0.713	0.046	0.179
STONE	4	0.597	0.034	0.102	STONE	4	0.807	0.141	0.297
STONE	8	0.634	0.039	0.088	STONE	8	0.868	0.133	0.482
STONE	16	0.682	0.078	0.176	STONE	16	0.952	0.653	0.790
SWEET	1	0.787	0.152	0.310	SWEET	1	0.962	0.565	0.824
SWEET	4	0.891	0.449	0.551	SWEET	4	0.998	0.950	0.988
SWEET	8	0.937	0.539	0.794	SWEET	8	**1.000**	**1.000**	**1.000**
SWEET	16	0.975	0.569	0.882	SWEET	16	**1.000**	**1.000**	**1.000**
SEAL	1	0.708	0.012	0.093	SEAL	1	0.696	0.012	0.067
SEAL	4	0.862	0.100	0.375	SEAL	4	0.813	0.050	0.325
SEAL	8	**0.941**	**0.600**	0.650	SEAL	8	0.921	0.400	0.600
SEAL	16	**1.000**	**1.000**	**1.000**	SEAL	16	0.991	0.900	0.900

we sample $N = 64$ candidate function bodies using temperature sampling with temperature 0.7, top-$p = 0.95$, and a maximum generation length of 256 tokens. Random projection vectors are generated from a fixed pivot seed, and secret bits are generated from a fixed secret seed.

Embedding and Verification. SEAL uses sequential median filtering during embedding. Starting from the filtered candidate pool, the method applies the keyed semantic channels one by one and keeps candidates that match the secret sign whenever the surviving set is nonempty. If no candidate survives a channel constraint, the method skips that constraint and continues with the current pool. At the end, SEAL selects the remaining candidate with the highest total agreement score. Verification reconstructs the reference pool using the same prompt, sampling configuration, seed, and filtering rule, then computes prompt-specific medians to recover the semantic signs.

5.2 Evaluation Metrics

Our main experiments evaluate controlled watermark detectability. Watermarked outputs are treated as positives and unwatermarked model outputs are treated as negatives. We report AUROC, TPR@1%FPR, and TPR@5%FPR, consistent with prior work Zhang et al. (2025); Feng et al. (2024); Wu et al. (2025). For aggregation, we group K generated outputs and combine their detector scores before computing the same metrics. We apply the same aggregation levels to SEAL, STONE Kim et al. (2026), and SWEET Lee et al. (2024). Our current experiments filter candidate pools with benchmark unit tests before watermark selection, so we report them as controlled detectability experiments rather than zero-shot Pass@1 or utility evaluations. This setup isolates whether SEAL can embed a recoverable watermark among functionally valid candidates. A more rigorous deployment setting, where the selector cannot use benchmark test outcomes and must rely only on available checks such as syntax validation, public tests, static analysis, or model-based quality filters, is left to future work.

5.3 Results and Discussion

As demonstrated in Tables 1 and 2, in the single-output setting, SEAL produces a moderate detection signal. This is expected because SEAL does not bias token probabilities during decoding or retrain the model. Instead, it introduces a semantic agreement bias through candidate selection. On CodeLlama-7B-Instruct Roziere et al. (2023), SEAL obtains an AUROC of 0.712 on HumanEval and 0.738 on MBPP at $K = 1$, outperforming STONE Kim et al. (2026) already. On StarCoder2-7B Lozhkov et al. (2024), SEAL obtains an AUROC of 0.708 on HumanEval and 0.696 on MBPP at $K = 1$.

We then observe that aggregation substantially strengthens SEAL detection, providing experimental proof for our Theorem 1. On CodeLlama-7B-Instruct Roziere et al. (2023) with HumanEval, SEAL improves from 0.712 AUROC at $K = 1$ to 0.875 at $K = 4$, 0.950 at $K = 8$, and 0.986 at $K = 16$. On CodeLlama-7B-Instruct Roziere et al. (2023) with MBPP, SEAL improves from 0.738 AUROC at

Table 3: LLM rewrite robustness results on both HumanEval and MBPP datasets.

		Qwen2.5-Coder-7B-Instruct		GPT-4o	
Dataset	K	Clean AUROC ↑	Rewrite AUROC ↑	Clean AUROC ↑	Rewrite AUROC ↑
HumanEval	1	0.712	0.688	0.728	0.619
HumanEval	4	0.875	0.848	0.905	0.743
HumanEval	8	0.950	0.927	0.969	0.856
HumanEval	16	**0.986**	**0.978**	**0.993**	**0.921**
MBPP	1	0.738	0.701	0.738	0.604
MBPP	4	0.907	0.857	0.907	0.708
MBPP	8	0.978	0.941	0.978	0.796
MBPP	16	**0.999**	**0.991**	**0.999**	**0.882**

$K = 1$ to 0.999 at $K = 16$. These results match our theoretical analysis: each function contributes a weak keyed semantic signal, and aggregation increases the detection statistic. The StarCoder2-7B Lozhkov et al. (2024) results show the same aggregation trend, but also show that SEAL is not uniformly the strongest single-output detector. SWEET is particularly strong on StarCoder2-7B with MBPP. SEAL instead becomes more competitive as the verifier aggregates multiple generated functions. This supports the intended use case of SEAL as an audit-level attribution method rather than a method that forces a large watermark signal into every individual snippet.

The $K = 16$ rows demonstrate the multi-function verification performance of each method, not single-snippet attribution. Larger K reduces variance in the aggregate detector score, but it also reduces the number of aggregate groups. For example, grouping HumanEval into sets of size 16 yields only about ten positive groups, so low-FPR TPR values can become coarse. We therefore report several values of K and focus on the trend from $K = 1$ to $K = 4$ to $K = 8$. Overall, our experiments show that SEAL produces a detectable semantic watermark through systematic candidate selection.

5.4 Robustness Evaluation

We evaluate robustness under formatting, variable renaming, rule-based rewriting, and LLM rewrites. SEAL retains the strongest post-attack detection on CodeLlama-7B HumanEval across all transformations. Under variable renaming and rule-based rewriting, SEAL achieves $K = 16$ AUROC of 0.901 and 0.923, while STONE drops to 0.524 and 0.576, and SWEET reaches 0.677 and 0.704. The same trend appears at single-output detection, where SEAL keeps higher AUROC and TPR@5%FPR than both baselines after each edit. These empirical results align with our Theorem 2. The measured semantic bit-flip rates remain below the $\epsilon < 0.5$ threshold: 0.175 for formatting, 0.345 for variable renaming, and 0.342 for rule-based rewriting. Full robustness under code transformation results table can be found in Appendix A.3. We additionally test full LLM-based rewrites (Qwen2.5-Coder-7B-Instruct and GPT-4o, following the SWEET Lee et al. (2024) and STONE Kim et al. (2026) settings) on HumanEval and MBPP (Table 3); SEAL loses only 0.008 AUROC under Qwen rewrites at $K = 16$ and remains at $0.921/0.882$ AUROC under the stronger GPT-4o rewrite. Thus, the watermark weakens under semantics-preserving edits but remains recoverable, especially when evidence is aggregated across generated functions.

6 Conclusion

We presented SEAL, a semantic-space watermarking framework for code generation that embeds a secret-key-dependent signal through candidate selection rather than token-level biasing or model modification. SEAL treats each generated function as the watermark-carrying unit, maps candidates into a code embedding space, and selects valid outputs whose semantic signs match a secret key. Our theoretical analysis shows that the detection signal strengthens through aggregation and remains positive under transformations that preserve most recovered semantic signs. Experiments on HumanEval, MBPP, and DS-1000 results show that SEAL achieves strong controlled detectability, improves substantially with aggregation, and remains robust under formatting, renaming, and rule-based rewrites. We discuss limitations and future directions in Appendix A.5, including prompt-conditioned verification, inference overhead, and extensions toward more realistic deployment settings.

LLM USAGE

Large language models assisted with writing and editing during paper preparation, primarily for grammar, clarity, and restructuring purposes. All technical contributions, including method design, experiments, core implementation, and conclusions, are our own. LLMs were not used to generate results or scientific claims.

ETHICS STATEMENT

SEAL is intended to provide evidence about the origin of model-generated code, which may help with auditing and attribution when prompts and generation records are available. Because a watermark is a statistical signal that can weaken after code edits, a detection result should be considered alongside those records and human judgment, particularly before making consequential decisions about authorship or misconduct. Verification also requires access to prompts and secret keys, so these materials should be protected in practical deployments. Our evaluation uses public programming benchmarks and model-generated code and does not involve human participants or private user data.

REPRODUCIBILITY STATEMENT

Sections 3.2–3.3 define the embedding and verification algorithms. Sections 5.1–5.2 describe the datasets, models, computing hardware, generation settings, candidate filtering, and evaluation metrics, including the use of benchmark tests in the controlled experiments. Section 4 states the assumptions behind the theoretical results, with proofs in Appendix A.1. Additional results and the full robustness evaluation appear in Appendices A.2–A.3.

REFERENCES

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

Di Cao, Shigang Liu, Jun Zhang, and Yang Xiang. Robustness analysis of watermarking techniques for llm-generated code. *IEEE Networking Letters*, 7(4):265–269, 2025. doi: 10.1109/LNET.2025.3616325.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Greta Dolcetti, Vincenzo Arceri, Eleonora Iotti, Sergio Maffeis, Agostino Cortesi, and Enea Zaffanella. Helping llms improve code generation using feedback from testing and static analysis. *Discover Artificial Intelligence*, 6(1):314, 2026.

Xianheng Feng, Jian Liu, Kui Ren, and Chun Chen. A certified robust watermark for large language models. *arXiv preprint arXiv:2409.19708*, 2024.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547, 2020.

Eva Giboulot and Teddy Furon. Watermax: breaking the llm watermark detectability-robustness-quality trade-off. *Advances in Neural Information Processing Systems*, 37:18848–18881, 2024.

Tianle Gu, Zongqi Wang, Kexin Huang, Yuanqi Yao, Xiangliang Zhang, Yujiu Yang, and Xiuying Chen. Invisible entropy: Towards safe and efficient low-entropy llm watermarking. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 6727–6744, 2025.

Zhimeng Guo, Huaisheng Zhu, Siyuan Xu, Hangfan Zhang, Teng Xiao, and Minhao Cheng. Optimizing token choice for code watermarking: An rl approach. *arXiv preprint arXiv:2508.11925*, 2025.

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

Jiahao Huo, Shuliang Liu, Bin Wang, Junyan Zhang, Yibo Yan, Aiwei Liu, Xuming Hu, and Mingxun Zhou. Pmark: Towards robust and distortion-free semantic-level watermarking with channel constraints. *arXiv preprint arXiv:2509.21057*, 2025.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark. *arXiv preprint arXiv:2410.00752*, 2024.

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

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026.

Mohammed F Kharma, Soohyeon Choi, Mohammad Alkhanafseh, and David Mohaisen. Security and quality in llm-generated code: A multi-language, multi-model analysis. *IEEE Transactions on Dependable and Secure Computing*, 2026.

Jungin Kim, Shinwoo Park, and Yo-Sub Han. Marking code without breaking it: Code watermarking for detecting llm-generated code. In *Findings of the Association for Computational Linguistics: EACL 2026*, pages 3990–4002, 2026.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. A watermark for large language models. In *International conference on machine learning*, pages 17061–17084. PMLR, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. 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)*, pages 4890–4911, 2024.

Boquan Li, Zirui Fu, Mengdi Zhang, Peixin Zhang, Jun Sun, and Xingmei Wang. Efficient and universal watermarking for llm-generated code detection. *arXiv preprint arXiv:2402.07518*, 2024.

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

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024.

Yijian Lu, Aiwei Liu, Dianzhi Yu, Jingjing Li, and Irwin King. An entropy-based text watermarking detection method. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11724–11735, 2024.

Nathan Rosenblum, Xiaojin Zhu, and Barton P Miller. Who wrote this code? identifying the authors of program binaries. In *European Symposium on Research in Computer Security*, pages 172–189. Springer, 2011.

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

Tom Sander, Pierre Fernandez, Saeed Mahloujifar, Alain Durmus, and Chuan Guo. Detecting benchmark contamination through watermarking. *arXiv preprint arXiv:2502.17259*, 2025.

Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. Codemark: Imperceptible watermarking for code datasets against neural code completion models. In *Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering*, pages 1561–1572, 2023.

Tarun Suresh, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Is the watermarking of llm-generated code robust? *arXiv preprint arXiv:2403.17983*, 2024.

Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. Domain adaptive code completion via language models and decoupled domain databases. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 421–433. IEEE, 2023.

Yihan Wu, Xuehao Cui, Ruibo Chen, and Heng Huang. Analyzing and evaluating unbiased language model watermark. *arXiv preprint arXiv:2509.24048*, 2025.

Xiaojun Xu, Yuanshun Yao, and Yang Liu. Learning to watermark llm-generated text via reinforcement learning. *arXiv preprint arXiv:2403.10553*, 2024.

Xianjun Yang, Liangming Pan, Xuandong Zhao, Haifeng Chen, Linda Ruth Petzold, William Yang Wang, and Wei Cheng. A survey on detection of llms-generated content. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 9786–9805, 2024.

Zhuohao Yu, Xingru Jiang, Weizheng Gu, Yidong Wang, Shikun Zhang, and Wei Ye. Saemark: Multi-bit llm watermarking with inference-time scaling. *arXiv preprint arXiv:2508.08211*, 2025.

Yu Zhang, Shuliang Liu, Xu Yang, and Xuming Hu. Catmark: A context-aware thresholding framework for robust cross-task watermarking in large language models. *arXiv preprint arXiv:2510.02342*, 2025.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2): 1–124, 2023a.

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

Xuandong Zhao, Sam Gunn, Miranda Christ, Jaiden Fairoze, Andres Fabrega, Nicholas Carlini, Sanjam Garg, Sanghyun Hong, Milad Nasr, Florian Tramer, et al. Sok: Watermarking for ai-generated content. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2621–2639. IEEE, 2025.

A Appendix

A.1 Theoretical Proofs

We present the following mathematical proofs for our two proposed theorems in the main paper.

Theorem 1. SEAL detection improves with aggregation.

Proof. Under the balanced median construction, each unwatermarked channel agrees with the secret sign with probability $1/2$. Assuming independent channel evidence across prompts and projection directions, the total number of agreements over K functions and m channels is a sum of Km Bernoulli random variables with success probability $1/2$. Hence,

$$S \mid H_0 \sim \text{Binomial}(Km, 1/2).$$

Under SEAL selection, each channel has agreement probability $1/2 + \delta$, so linearity of expectation gives

$$\mathbb{E}_{H_1}[S] = \sum_{k=1}^{K} \sum_{j=1}^{m} \Pr\left[q_j(y_k; x_k) = s_{k,j} \mid H_1\right] = Km\left(\frac{1}{2} + \delta\right).$$

The standardized statistic subtracts the null mean $Km/2$ and divides by the null standard deviation $\sqrt{Km/4}$. Therefore,

$$\mathbb{E}_{H_1}[Z] = \frac{Km(\frac{1}{2} + \delta) - Km/2}{\sqrt{Km/4}} = \frac{Km\delta}{\sqrt{Km}/2} = 2\delta\sqrt{Km}.$$

This proves the stated scaling law. □

Theorem 2. SEAL remains detectable as long as the attacker flips fewer than 50% of the semantic watermark signs.

Proof. After the attack, the recovered sign agrees with the secret sign in two cases. First, the original sign agrees with the secret sign and the attack does not flip it. Second, the original sign disagrees with the secret sign and the attack flips it. Therefore,

$$\Pr[\tilde{s}_j = s_j] = (1 - \epsilon)\Pr[\hat{s}_j = s_j] + \epsilon \Pr[\hat{s}_j \neq s_j] \tag{15}$$

$$= (1 - \epsilon)\left(\frac{1}{2} + \delta\right) + \epsilon\left(\frac{1}{2} - \delta\right) \tag{16}$$

$$= \frac{1}{2} + \delta(1 - 2\epsilon). \tag{17}$$

Applying the detection scaling from Theorem 1 with the reduced margin

$$\delta_{\text{attack}} = \delta(1 - 2\epsilon)$$

gives

$$\mathbb{E}[Z_{\text{attack}}] = 2\delta(1 - 2\epsilon)\sqrt{Km}.$$

For $\epsilon < 1/2$, this value is positive, so the transformed code still retains above-chance watermark agreement.

Geometric interpretation. The bit-flip probability ϵ depends on how far a transformation moves the code in embedding space. Let

$$\mathbf{z}_y = f_{\text{enc}}(y), \qquad \mathbf{z}_{\tilde{y}} = f_{\text{enc}}(\tilde{y}), \qquad \Delta\mathbf{z} = \mathbf{z}_{\tilde{y}} - \mathbf{z}_y.$$

A sign flip in channel j occurs only if the transformation crosses the median boundary:

$$\text{sign}\left(\langle \mathbf{z}_y, \mathbf{r}_j \rangle - \mu_j(x)\right) \neq \text{sign}\left(\langle \mathbf{z}_{\tilde{y}}, \mathbf{r}_j \rangle - \mu_j(x)\right).$$

Equivalently, a flip can occur only when

$$\left|\langle \mathbf{z}_y, \mathbf{r}_j \rangle - \mu_j(x)\right| \leq \left|\langle \Delta\mathbf{z}, \mathbf{r}_j \rangle\right|.$$

Thus, an edit is most likely to remove watermark evidence when the original embedding lies close to a median boundary. If a transformation preserves the semantic representation, then $\|\Delta\mathbf{z}\|$ is small, and only a limited number of channel boundaries are crossed. This explains why formatting changes, renaming, and moderate semantics-preserving rewrites should degrade SEAL gradually rather than erase it abruptly.

Table 4: Partial DS1000 detection results for SEAL with CodeLlama-7B-Instruct. We evaluate a subset of DS1000 using $m = 8$ semantic channels and report single-output detection and PMark-style aggregation over K generated functions. The run contains 164 evaluated prompts.

Aggregation	Bit Acc. ↑	AUROC ↑	TPR@1%FPR ↑	TPR@5%FPR ↑
$K = 1$	0.637	0.718	0.012	0.189
$K = 2$	0.637	0.808	0.037	0.232
$K = 4$	0.637	0.862	0.244	0.512
$K = 8$	0.639	0.927	0.600	0.800
$K = 16$	0.639	0.971	0.800	0.800

Table 5: Robustness comparison under code transformations on CodeLlama-7B HumanEval. We report detection performance for STONE, SWEET, and SEAL before and after transformations. For all methods, K=16 AUROC denotes aggregated verification over groups of 16 outputs.

	STONE			SWEET			SEAL		
Attack	AUROC ↑	TPR@5% ↑	K=16 AUROC ↑	AUROC ↑	TPR@5% ↑	K=16 AUROC ↑	AUROC ↑	TPR@5% ↑	K=16 AUROC ↑
None	0.547	0.070	0.596	0.620	0.140	0.741	**0.723**	**0.403**	**0.997**
Formatting	0.542	0.071	0.583	0.599	0.118	0.704	**0.687**	**0.336**	**0.987**
Rename	0.515	0.035	0.524	0.580	0.089	0.677	**0.611**	**0.262**	**0.901**
Rule rewrite	0.542	0.063	0.576	0.599	0.111	0.704	**0.632**	**0.248**	**0.923**

A.2 More Experimental Results

We provide additional experimental results on DS-1000 Lai et al. (2023) of our SEAL method. Table 4 reports preliminary SEAL results on a randomly selected portion of the DS-1000 Lai et al. (2023) dataset. We can observe that the detection trend is consistent with prior HumanEval and MBPP results: single-output detection is moderate, while aggregation substantially improves low-FPR detection. AUROC increases from 0.718 at $K = 1$ to 0.971 at $K = 16$, and TPR@1%FPR increases from 0.012 to 0.800. These results suggest that SEAL's aggregation behavior extends beyond function-synthesis benchmarks to data-science-oriented code generation tasks. We also provide some clean and watermarked code examples from all three datasets in Figure 2.

A.3 Full Robustness Experiments

We provide full robustness results under code transformations on CodeLlama-7B HumanEval in Table 5, and results for CodeLlama-7B and StarCoder2-7B in Table 6. For each selected watermarked program, we apply three semantics-preserving transformations: formatting, variable renaming, and rule-based rewriting. After each transformation, we re-embed the transformed code, recover semantic signs using the original prompt-specific medians, and measure syntax validity, bit accuracy, empirical flip rate ϵ, and post-attack detectability. The flip rate directly instantiates the perturbation term in Theorem 2, which predicts that the watermark signal remains positive when $\epsilon < 0.5$.

On CodeLlama-7B, SEAL remains robust across all tested transformations. Formatting has the smallest effect, increasing the flip rate from 0.086 to 0.175, while keeping $K = 16$ AUROC high at 0.987. Variable renaming and rule-based rewriting are stronger perturbations, with flip rates of 0.345 and 0.342, respectively. Although single-output AUROC drops to 0.611 and 0.632, aggregated verification remains strong, with $K = 16$ AUROC values of 0.901 and 0.923. This shows that SEAL weakens under stronger edits but remains recoverable in the intended multi-output setting.

StarCoder2-7B presents a harder boundary case. All flip rates remain below 0.5, so the expected watermark signal is not fully erased, but the margin after transformation is small. Formatting, variable renaming, and rule-based rewriting reduce bit accuracy to 0.497, close to chance, and the corresponding $K = 16$ AUROC values drop to 0.671, 0.653, and 0.669. These results match the theorem's interpretation: preserving more than half of the semantic signs is enough to keep a positive expected signal, but practical detectability depends on how far the flip rate remains below 0.5 and how stable the encoder representation is under transformation.

SEAL is designed to support attribution of LLM-generated code. This can help model providers, benchmark maintainers, and software organizations audit whether code was produced by a specific watermarked model. In settings where code provenance matters, such as education, enterprise

Table 6: Full robustness results of SEAL. We report syntax validity, post-attack bit accuracy, empirical bit-flip rate ϵ, and detection performance at both single-output and aggregated verification. Theorem 2 predicts that the watermark signal remains positive when $\epsilon < 0.5$.

Model	Attack	Syntax Valid ↑	Bit Acc. ↑	Flip Rate ϵ ↓	AUROC ↑	TPR@5%FPR ↑	$K = 16$ AUROC ↑
CodeLlama-7B	None	1.000	0.638	0.086	0.723	0.403	0.997
CodeLlama-7B	Formatting	1.000	0.612	0.175	0.687	0.336	0.987
CodeLlama-7B	Variable rename	1.000	0.565	0.345	0.611	0.262	0.901
CodeLlama-7B	Rule rewrite	1.000	0.577	0.342	0.632	0.248	0.923
StarCoder2-7B	None	0.988	0.588	0.104	0.677	0.293	0.983
StarCoder2-7B	Formatting	0.988	0.497	0.368	0.540	0.150	0.671
StarCoder2-7B	Variable rename	0.988	0.497	0.375	0.539	0.143	0.653
StarCoder2-7B	Rule rewrite	0.988	0.497	0.370	0.539	0.143	0.669

DS-1000

Clean Example

```
import pandas as pd

df = pd.DataFrame({
    'product': [1179160, 1066490, 1148126, 1069104, 1069105,
                1160330, 1069098, 1077784, 1193369, 1179741],
    'score': [0.424654, 0.424509, 0.422207, 0.420455, 0.414603,
              0.168784, 0.168749, 0.168738, 0.168703, 0.168684]
})
products = [1066490, 1077784]

mask = df['product'].isin(products)
df.loc[mask, 'score'] = df.loc[mask, 'score'] * 10

print(df)
```

Watermarked Example

```
import pandas as pd

df = pd.DataFrame({
    'product': [1179160, 1066490, 1148126, 1069104, 1069105,
                1160330, 1069098, 1077784, 1193369, 1179741],
    'score': [0.424654, 0.424509, 0.422207, 0.420455, 0.414603,
              0.168784, 0.168749, 0.168738, 0.168703, 0.168684]
})
products = [1066490, 1077784]

df.loc[df['product'].isin(products), 'score'] = \
    df.loc[df['product'].isin(products), 'score'] * 10
print(df)
```

HumanEval

Clean Example

```
whole_note, half_note, quarter_note = 4, 2, 1
notes = music_string.split()
notes_list = []

for note in notes:
    if note == 'o':
        notes_list.append(whole_note)
    elif note == 'o|':
        notes_list.append(half_note)
    elif note == '.|':
        notes_list.append(quarter_note)

return notes_list
```

Watermarked Example

```
music_string = music_string.strip()
notes = music_string.split()
beats = []

for note in notes:
    if note == 'o':
        beats.append(4)
    elif note == 'o|':
        beats.append(2)
    elif note == '.|':
        beats.append(1)
    else:
        raise ValueError("Invalid note in music string")

return beats
```

MBPP

Clean Example

```
result = []
current = ""

for char in string:
    if char.islower():
        current += char
    else:
        if current:
            result.append(current)
            current = ""

if current:
    result.append(current)

return result
```

Watermarked Example

```
result = []
current_word = ""
for char in string:
    if char.islower():
        current_word += char
    else:
        if current_word:
            result.append(current_word)
            current_word = ""
if current_word:
    result.append(current_word)
return result
```

Figure 2: Qualitative examples of clean and watermarked programs. For each dataset, we show one clean surviving candidate and the SEAL-selected watermarked candidate for the same prompt.

software development, and benchmark evaluation, a reliable watermark can provide useful evidence without requiring changes to the base model or its output token distributions.

At the same time, watermarking should not be treated as a complete solution for code provenance. Detection results are statistical, and they should not be used as the only basis for high-stakes decisions about authorship, misconduct, or ownership. SEAL also assumes access to prompt-conditioned verification information, so misuse or poor logging can weaken its reliability. More broadly, any watermarking system can create incentives for removal attacks, laundering, or adversarial rewriting. We therefore view SEAL as one component of a larger provenance pipeline, alongside logging, version control, signed generation records, and human review.

A.4 Computational overhead and runtime analysis.

SEAL incurs additional inference cost because it samples and filters multiple candidate programs before selecting a watermarked output. Candidate generation, however, can be parallelized. Table 7 reports generation runtime for 64 candidates at different concurrency levels, normalized to the runtime for one candidate. At concurrency $C = 64$, generation takes $2.30\times$ the single-candidate runtime on HumanEval and $4.21\times$ on MBPP, corresponding to speedups of $21.37\times$ and $19.05\times$ over generating 64 candidates serially. These measurements cover candidate generation; filtering, encoding, and watermark selection introduce additional costs.

Table 7: Parallel candidate-generation runtime. Runtime is normalized to a single-candidate, single-pass baseline. Speedup is relative to generating the same 64 candidates with concurrency $C = 1$. Bracketed values are 95% intervals.

Benchmark	Candidates	Concurrency C	Speedup vs. $C = 1$	Normalized runtime
HumanEval	1	1	—	$1.00\times$
HumanEval	64	1	$1.00\times$ [$1.00\times$, $1.00\times$]	$49.24\times$ [$35.90\times$, $68.70\times$]
HumanEval	64	8	$6.38\times$ [$5.25\times$, $7.06\times$]	$7.72\times$ [$5.35\times$, $11.09\times$]
HumanEval	64	16	$11.64\times$ [$11.00\times$, $12.22\times$]	$4.23\times$ [$3.04\times$, $5.98\times$]
HumanEval	64	32	$15.39\times$ [$13.30\times$, $17.48\times$]	$3.20\times$ [$2.28\times$, $4.58\times$]
HumanEval	64	64	$\mathbf{21.37}\times$ [$\mathbf{18.07}\times$, $\mathbf{25.49}\times$]	$\mathbf{2.30}\times$ [$\mathbf{1.70}\times$, $\mathbf{3.27}\times$]
MBPP	1	1	—	$1.00\times$
MBPP	64	1	$1.00\times$ [$1.00\times$, $1.00\times$]	$80.12\times$ [$55.95\times$, $117.16\times$]
MBPP	64	8	$6.86\times$ [$6.60\times$, $7.08\times$]	$11.67\times$ [$7.99\times$, $17.39\times$]
MBPP	64	16	$11.25\times$ [$10.54\times$, $11.92\times$]	$7.12\times$ [$4.94\times$, $10.68\times$]
MBPP	64	32	$16.01\times$ [$14.80\times$, $17.27\times$]	$5.00\times$ [$3.40\times$, $7.50\times$]
MBPP	64	64	$\mathbf{19.05}\times$ [$\mathbf{17.21}\times$, $\mathbf{21.27}\times$]	$\mathbf{4.21}\times$ [$\mathbf{2.90}\times$, $\mathbf{6.33}\times$]

A.5 Limitations and Future Work

SEAL has two main limitations. First, SEAL uses prompt-conditioned verification. The verifier needs the original prompt, secret key, encoder, and sampling procedure to reconstruct the prompt-specific reference distribution used during watermarking. This setting is natural for code generation services, benchmark auditing, and enterprise deployments where prompts and outputs are logged, but it does not provide fully prompt-free attribution. A future prompt-free extension can be implemented through storing verification metadata, such as prompt-specific medians, or replace prompt-specific thresholds with global calibration thresholds.

Second, SEAL also has higher inference cost than methods that watermark a single generation pass. It generates multiple candidates, filters them, encodes the surviving programs, and selects an output. Table 7 shows that parallel generation substantially reduces the latency of candidate sampling, but the computational overhead is still present. The current method is therefore more suitable for offline generation or attribution settings where the additional compute is acceptable than for applications requiring low-latency responses.

Future work could reduce inference cost through smaller candidate pools, lighter code encoders, cached embeddings, or adaptive sampling. Verification could also be simplified by storing prompt-specific medians or learning calibrated global thresholds, although both approaches would need to be evaluated when the original candidate pool cannot be reconstructed.