SEW: STYLE-ENCODED WATERMARKING OF LLM-GENERATED CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

Code watermarking supports provenance tracking for code generated by LLMs. Modifying token selection to embed watermarks as an LLM generates code can create a trade-off between detectability and functional correctness. Other methods instead watermark completed code using predefined transformations or trained neural models. Recurring patterns can make watermark choices predictable across programs, while treating patterns common in unwatermarked code as watermark evidence can cause false detections. We therefore introduce **SEW**, which embeds and detects watermarks in already generated code through three components: (i) code style rules collected from style guides and transformation rules, with style choices determined by a secret key and each program's structural context; (ii) style-preference calibration, which evaluates watermark evidence using style probabilities estimated from human-written code; and (iii) context-aware style aggregation, which combines evidence from structurally matching locations assigned the same code style choice, preventing repeated applications of that choice from inflating watermark evidence. On CodeContests across three LLMs and three programming languages, SEW achieves a mean relative improvement of 11.90% in TPR@FPR5% over the baselines and is robust to four non-LLM code-editing attacks, with only a 0.94% mean relative decrease. Our code is available at `https://anonymous.4open.science/r/Code_Watermark`.

1 INTRODUCTION

The use of large language models (LLMs) for code generation creates a need to verify the provenance of their outputs. Code watermarking addresses this need by embedding detectable signals that support subsequent attribution (Lee et al., 2024; Yang et al., 2024). These signals must distinguish marked code from unmarked code without compromising program behavior. A useful code watermark must therefore combine reliable detection with functional correctness.

Watermarks can be embedded by modifying token selection while an LLM generates code or by transforming the completed code. Modifying token probabilities in low-entropy code can create a trade-off between detectability and functional correctness, motivating entropy- and syntax-aware strategies (Lee et al., 2024; Kim et al., 2026). For completed code, existing approaches use fixed transformations or trained neural embedding and extraction models (Li et al., 2026; Yang et al., 2024; Zhang et al., 2025b). Fixed transformations reuse predefined choices, while learned transformation selectors can also favor recurring patterns. When the same choices recur across programs, the resulting patterns can become easier to anticipate and imitate.

We therefore introduce **SEW**, **S**tyle-**E**ncoded **W**atermarking, which embeds and detects watermarks in code already generated by an LLM. SEW integrates three components: (i) *code style rules*, constructed from style guides (van Rossum et al., 2001; Google, n.d.) and existing code transformations (Wang et al., 2023; Yang et al., 2024; Li et al., 2026). Each rule supports conversion between two syntax or formatting variants, with SEW selecting the variant using a secret key and each program's structural context; (ii) *style-preference calibration*, which evaluates watermark evidence using probabilities estimated from code style patterns observed in human-written code, accounting for patterns that also occur without watermarking; and (iii) *context-aware style aggregation*, which combines evidence from structurally matching locations assigned the same code style choice, preventing repeated applications of that choice from inflating watermark evidence.

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.

Table 1: Comparison of SEW and existing code watermarking. ✓: criterion met; ✗: not met.

Method	Embedding stage	Model-free detection	Pattern-resistant	Function-preserving
KGW (Kirchenbauer et al., 2023)	During code generation	✓	✓	✗
Unigram (Zhao et al., 2024)		✓	✗	✗
SWEET (Lee et al., 2024)		✗	✓	✗
STA-1 (Mao et al., 2025)		✓	✓	✗
STONE (Kim et al., 2026)		✓	✓	✗
SrcMarker (Yang et al., 2024)	After code generation	✗	✗	✗
RoSeMary (Zhang et al., 2025b)		✗	✗	✗
ACW (Li et al., 2026)		✓	✗	✗
SEW (ours)		✓	✓	✓

Table 1 compares SEW and existing code watermarking approaches regarding model-free detection, pattern resistance, and functional preservation. Model-free detection denotes detection without running the generating LLM or a separately trained extraction model. Pattern resistance concerns whether an adversary can exploit fixed or recurring patterns in collected watermarked outputs to infer watermark choices in other programs. Functional preservation requires maintaining the pass@1 of the corresponding unwatermarked outputs under matched evaluation conditions.

Figure 1 compares TPR@FPR5% and pass@1 on CodeContests (Li et al., 2022a), averaged across three LLMs and three programming languages. Filled markers denote watermark insertion into completed code, whereas open markers denote insertion while the LLM generates code. The dashed line provides the unwatermarked pass@1 reference for assessing functional preservation when watermarking completed code. The evaluated baselines have lower detection rates than SEW, and learning-based transformations of completed code also reduce pass@1. SEW achieves a mean TPR@FPR5% of 98.71% while preserving the pass@1 of the corresponding unwatermarked code across all nine model–language settings. SEW therefore combines high detectability with preserved benchmark functional correctness.

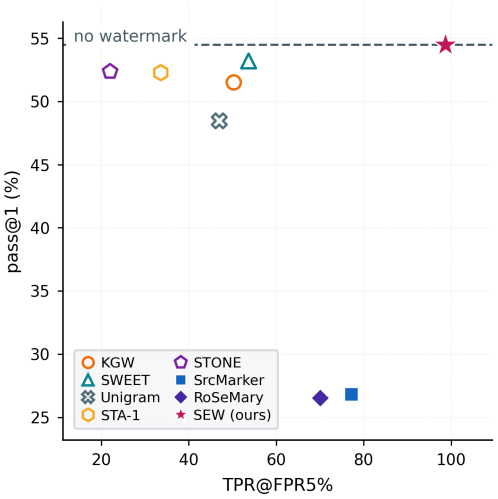

Figure 1: Detection vs. pass@1.

We make three main contributions:

- **Structure-aware code style watermarking.** We construct a code style rule library and select styles using a secret key and structural context without modifying LLM decoding.
- **Calibrated, context-aware style detection.** We combine style-preference calibration with context-aware style aggregation to evaluate code style patterns as watermark evidence.
- **Empirical validation across models and languages.** SEW outperforms baselines in TPR@FPR5% on CodeContests, preserves pass@1, and resists five code-editing attacks.

2 Related Work

LLM watermarking. Watermarking during LLM generation requires decoding access. KGW favors keyed green tokens and detects their overrepresentation (Kirchenbauer et al., 2023), whereas Unigram uses a fixed vocabulary partition with robustness guarantees under bounded text edits (Zhao et al., 2024). STA-1 preserves the unwatermarked next-token distribution in expectation while limiting the risk of low-probability outputs in low-entropy contexts (Mao et al., 2025).

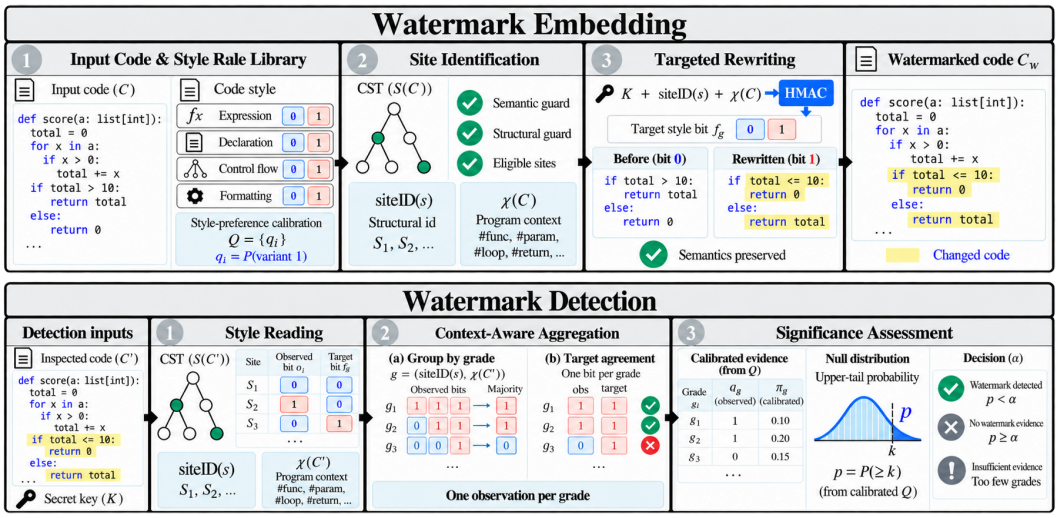

Figure 2: An overview of the SEW pipeline.

Code watermarking. Code watermarking accounts for syntax and functional requirements. SWEET excludes low-entropy positions from embedding and detection (Lee et al., 2024), whereas STONE restricts embedding to non-syntactic tokens (Kim et al., 2026). Both require decoding access. Post-hoc methods transform completed programs. SrcMarker requires trained neural embedding and extraction modules (Yang et al., 2024). RoSeMary jointly trains a CodeT5-based embedding module and watermark extractor (Zhang et al., 2025b). ACW avoids model training through predefined, idempotent transformations (Li et al., 2026), but its recurring style choices are susceptible to inference from collected watermarked programs. SEW instead applies structure-conditioned, secret-key-guided code style transformations without decoding access or neural model training.

Code style. Style documentation (Sun Microsystems, 1999; van Rossum et al., 2001; Brandl & Storchaka, 2016; Langa & contributors to Black, 2026; Google, n.d.) and tool rules describe naming, formatting, and syntactic conventions. Studies examine coding conventions (Allamanis et al., 2014), authorship attribution and robustness (Caliskan-Islam et al., 2015; Quiring et al., 2019; Li et al., 2022b), style transfer and generation (Ting et al., 2023; Dai et al., 2024; Chen et al., 2025; Zhang et al., 2025a), style inconsistencies in LLM-generated code (Wang et al., 2025), and semantics-preserving transformations (Chakraborty et al., 2022; Sun et al., 2023; Li et al., 2026). SEW aims to embed watermarks using context-appropriate syntax and formatting alternatives without conspicuous stylistic changes.

3 METHOD

Given completed code C and a secret key K, SEW embeds a watermark through code style choices and detects it from inspected code C' using the same key. SEW assigns key-dependent style choices according to structural context and applies them consistently to matching sites within each program. This design aims to embed watermarks without introducing conspicuous stylistic inconsistencies. Since these styles can also occur without watermarking, detection must account for natural style preferences and repeated applications of the same choice. SEW therefore combines structure-aware style selection with calibrated, context-aware detection. (i) The style rule library defines binary syntax and formatting alternatives, their applicability conditions, and estimated natural variant probabilities. (ii) Watermark embedding identifies eligible sites and selects variants using the key and structural context. (iii) Watermark detection reads the inspected styles, aggregates repeated choices, and evaluates target agreement against the estimated natural probabilities. Neither embedding nor detection requires access to the generation model. Figure 2 presents the SEW pipeline.

Table 2: Code style choices used by SEW, grouped by kind.

Kind	Style choices
Expression	comparison direction, operand order, redundant parentheses, emptiness and size tests, default arguments, literal forms
Declaration	augmented, tuple and chained assignment, multiple declaration, increment form, `final`/`const` locals
Control flow	branch order, conditional expression, braces on a single statement, loop forms, return forms, comprehensions
Formatting	operator/keyword spacing, bracket indent, blank lines, final newline

3.1 Style Rule Library

The style rule library represents code style alternatives as binary choices for watermark embedding and detection. A style pair must support conversion in both directions, but its applicability and natural occurrence frequencies depend on the code. We therefore constructed the library through code style collection, rule-specific applicability conditions, and style-preference calibration.

Code Style Collection. We manually collected code style alternatives from seven style guides, linter and refactoring rule sets, and thirteen prior studies. We organized these alternatives into Python, Java, and C++ catalogs covering assignment forms, comparison direction, parenthesization, and operator spacing. For language ℓ, let $\mathcal{R}_\ell$ denote its rule library. Each rule $r \in \mathcal{R}_\ell$ defines two style variants labeled 0 and 1. Rule-specific applicability checks cover type, side-effect, and structural conditions and are shared by embedding and detection. When applicable, each rule supports bidirectional conversion to realize the target bit determined by the secret key and structural context, rather than normalize code toward a single preferred style. The library contains 29 Python, 22 Java, and 19 C++ rules. Table 2 summarizes the rule categories. Appendix F provides the full rule list and examples of applicability conditions.

Style-Preference Calibration. The two variants of a style rule need not occur equally often in unwatermarked code. We estimated their natural frequencies from a separate corpus of user-posted LeetCode solutions (kaysss, 2025), without using the CodeContests evaluation solutions. We applied the fixed rules and their applicability checks to this corpus and counted eligible occurrences of each variant. For rule r, let $n_{r,0}$ and $n_{r,1}$ denote these counts. For rules with observed occurrences, the empirical probability of variant 1 is $\widehat{q}_r = n_{r,1}/(n_{r,0} + n_{r,1})$. We obtained q_r by clipping the estimates to $[0.05, 0.95]$, with $q_r = 1/2$ for rules with fewer than 30 observations or unstable estimates. The resulting table $Q = \{q_r\}_{r \in \mathcal{R}_\ell}$ provides the natural-style baseline for detection. At a site, the estimated chance of matching a target of 1 is q_r, whereas that of matching a target of 0 is $1 - q_r$. A commonly used target variant can therefore match even without watermarking. By using its estimated natural probability instead of $1/2$, the detector assigns less watermark evidence to such expected matches, limiting false detections caused by naturally frequent code styles. Calibration affects detection only; it does not change the secret-key targets or the embedding procedure. Appendix C reports calibration corpus sizes and probability-estimation times.

3.2 Watermark Embedding

Given code C, language ℓ, and key K, SEW embeds a watermark through structure-aware code style choices without modifying generation. It analyzes the code to obtain its program context and identifies eligible style sites from the concrete syntax tree. For each site, SEW combines K, the program context, and its structural identifier to determine a binary target style. Sites with the same structural identifier and context receive the same target, so SEW applies style choices consistently to structurally matching locations rather than transforming matching patterns uniformly.

Site Identification. SEW parses the code into a concrete syntax tree (CST) and performs scope and def-use analysis to identify occurrences of code style rules. An occurrence enters the eligible site collection $\mathcal{S}(C)$ only when it satisfies the rule-specific semantic and structural conditions for interchanging its two variants. For example, in Python, the augmented-assignment rule converts between `x += y` and `x = x + y` only when `x` is inferred to be numeric; list updates are excluded

because in-place modification differs from reassignment. The same eligibility checks govern embedding and detection. Each eligible site s receives a structural identifier $\mathrm{siteID}(s)$ comprising the style rule, statement type, parent and grandparent node types, and block nesting depth. This identifier is independent of line numbers and identifier names. Site selection is public and deterministic; the secret key determines the target style rather than which sites are selected.

Context-Conditioned Embedding. SEW summarizes the program context $\chi(C)$ using four structural counts: functions, total parameters, loops, and return statements. For each eligible site s, SEW combines the secret key K, the site's structural identifier, and the program context to compute

$$d_s = \text{HMAC-SHA256}\big(K,\ \mathrm{siteID}(s) \,\|\, \chi(C)\big), \qquad t_s = d_s[0] \bmod 2 \in \{0, 1\}.$$

Here, $\|$ denotes concatenation and $d_s[0]$ is the first byte of the HMAC output. The resulting bit t_s determines which of the two style variants is used at site s. SEW reads the current variant and rewrites the site only when its value differs from t_s. Sites with the same structural identifier and context receive the same target under a fixed key, enforcing consistent style choices at structurally matching locations. Conversely, the same style rule can receive different targets across programs with different contexts. Thus, knowing the public rule alone does not determine its target.

3.3 Watermark Detection

Given inspected code C' and the secret key K, SEW reconstructs the watermark evidence without a reference copy or embedding metadata. It recovers eligible style sites, recomputes their target bits, and aggregates repeated observations with the same structural identifier and program context before statistical testing. The final test evaluates whether the observed target agreement is unlikely under calibrated natural style preferences. It does not use the generation model.

Style Reading and Context-Aware Aggregation. SEW applies the same CST parsing, scope and def-use analysis, and applicability conditions used during embedding to recover $\mathcal{S}(C')$. For each recovered site s, it reads the observed style value $b_s \in \{0, 1\}$ and recomputes t_s from K, $\mathrm{siteID}(s)$, and $\chi(C')$. Sites with the same structural identifier and program context are grouped into a grade $g = (\mathrm{siteID}(s), \chi(C'))$. Because all sites in a grade share the same target, counting each occurrence separately would repeatedly count the same style choice. Let $\mathcal{G}$ denote the set of grades and t_g their shared target. SEW summarizes the observed variants in each grade by a majority value b_g, assigning $b_g = 1$ in case of a tie, and records target agreement as

$$a_g = \mathbf{1}[b_g = t_g], \qquad k = \sum_{g \in \mathcal{G}} a_g.$$

Each grade therefore contributes one observation, preventing repeated applications of the same code style choice from inflating watermark evidence.

Calibrated Statistical Test. Code style variants need not occur equally often in unwatermarked code. SEW therefore uses the calibration table Q estimated from human-written code to model target agreement under the null hypothesis. Using the rule-specific probabilities in Q, let π_g denote the null probability that grade g agrees with its target. Since each grade contributes a binary agreement observation, SEW models its null agreement as $Z_g \sim \mathrm{Bernoulli}(\pi_g)$. Assuming independent grades, SEW computes the tail probability as

$$p_{\text{point}} = \Pr\left[\sum_{g \in \mathcal{G}} Z_g \geq k\right].$$

This test uses all available grades. Because formatting rules capture code properties that can vary independently of program structure, SEW also applies the same test using only grades from syntax rules, so detection does not rely solely on formatting-rule evidence. It combines the resulting p_{syn} with p_{point} using a Bonferroni correction, giving $p = \min\{1, 2\min(p_{\text{point}}, p_{\text{syn}})\}$. Smaller values of p indicate stronger evidence against the calibrated null hypothesis.

Table 3: Detection and functional correctness. T@5: TPR@FPR5%; P@1: pass@1.

			Python			Java			C++		
Model	Method	Type	T@5	AUROC	P@1	T@5	AUROC	P@1	T@5	AUROC	P@1
Qwen3.5	Vanilla	–	–	–	41.43	–	–	36.05	–	–	36.67
	KGW	Pre-hoc	91.43	97.24	38.57	81.63	96.67	28.57	86.00	**98.72**	32.67
	SWEET	Pre-hoc	86.43	94.39	**50.00**	79.59	91.12	27.21	86.67	94.29	32.67
	Unigram	Pre-hoc	86.43	96.05	38.57	82.31	93.65	24.49	70.00	85.51	26.67
	STONE	Pre-hoc	45.00	67.33	45.00	23.81	66.85	25.85	20.00	43.33	31.33
	STA-1	Pre-hoc	75.71	93.89	48.57	51.70	85.41	26.53	83.33	96.05	34.00
	SrcMarker	Post-hoc	89.29	98.75	40.00	40.82	91.95	5.44	54.67	79.72	10.00
	RoSeMary	Post-hoc	98.57	98.21	40.00	40.14	94.70	5.44	80.00	89.32	8.67
	ACW	Post-hoc	89.29	94.64	40.00	–	–	–	–	–	–
	SEW (ours)		**100**	**100**	41.43	**96.60**	**97.37**	**36.05**	**98.00**	98.43	**36.67**
gemma-4	Vanilla	–	–	–	87.57	–	–	78.92	–	–	81.63
	KGW	Pre-hoc	33.14	74.29	84.62	39.46	77.38	77.84	65.31	90.95	77.04
	SWEET	Pre-hoc	62.72	88.16	82.84	24.86	71.98	76.22	66.33	91.99	80.61
	Unigram	Pre-hoc	68.05	95.84	84.62	24.86	71.11	72.43	64.29	92.84	77.55
	STONE	Pre-hoc	33.14	64.47	**87.57**	12.97	65.77	**78.92**	23.47	71.17	**82.14**
	STA-1	Pre-hoc	7.10	33.89	82.84	3.24	51.61	77.84	9.18	36.92	80.61
	SrcMarker	Post-hoc	90.53	97.64	86.98	86.49	94.39	13.51	89.29	91.96	14.29
	RoSeMary	Post-hoc	97.04	97.07	85.80	25.41	95.47	13.51	90.82	95.06	14.29
	ACW	Post-hoc	91.72	95.86	84.62	–	–	–	–	–	–
	SEW (ours)		**100**	**100**	**87.57**	**100**	**100**	**78.92**	**98.98**	**99.20**	81.63
gpt-oss	Vanilla	–	–	–	43.88	–	–	41.18	–	–	42.92
	KGW	Pre-hoc	20.41	70.14	38.78	8.82	42.87	45.10	26.61	63.94	40.34
	SWEET	Pre-hoc	30.61	72.98	**45.41**	19.12	66.57	**46.08**	27.04	75.44	37.77
	Unigram	Pre-hoc	17.86	75.02	39.29	2.45	49.34	37.75	6.87	35.03	34.76
	STONE	Pre-hoc	6.12	55.19	39.80	6.37	56.24	41.18	27.47	79.87	39.48
	STA-1	Pre-hoc	22.96	70.99	42.35	7.84	47.43	38.24	41.63	88.19	39.48
	SrcMarker	Post-hoc	90.82	97.18	43.37	87.25	94.40	16.67	65.67	72.98	11.16
	RoSeMary	Post-hoc	97.96	96.68	43.37	27.45	96.22	16.18	72.96	83.94	11.59
	ACW	Post-hoc	89.29	94.64	43.37	–	–	–	–	–	–
	SEW (ours)		**98.47**	**98.91**	43.88	**98.04**	**98.51**	41.18	**98.28**	**98.65**	**42.92**

4 Experimental Setup

4.1 Datasets and Evaluation Metrics

Datasets. We used the validation and test splits of CodeContests (Li et al., 2022a) for Python, Java, and C++. We refer to unwatermarked model outputs as Vanilla and excluded problems whose Vanilla output did not complete an evaluable program within the generation token budget. Evaluation sets differ across models but are shared across methods within each model–language setting. Appendix B provides the filtering procedure and problem counts.

Evaluation Metrics. We evaluate detection with TPR at 5% FPR (TPR@FPR5%) and AUROC, which measure performance at a fixed false-positive rate and across thresholds, respectively. Functional correctness is measured by pass@1, the fraction of CodeContests programs that pass all tests.

4.2 Evaluated Models and Baselines

Evaluated Models. We evaluated three open-weight LLMs: Qwen3.5-9B (Qwen Team, 2026), gemma-4-12B-it (El Abd et al., 2026), and gpt-oss-20b (OpenAI, 2025), denoted Qwen3.5, gemma-4, and gpt-oss, respectively. Each generated Python, Java, and C++ solutions, yielding nine model–language settings. Qwen3-Coder-30B-A3B-Instruct performed the LLM rewriting attack.

Baseline Strategies. We compare SEW against five token-level watermarking baselines, KGW (Kirchenbauer et al., 2023), SWEET (Lee et al., 2024), Unigram (Zhao et al., 2024), STONE (Kim et al., 2026), and STA-1 (Mao et al., 2025), and three post-hoc baselines, ACW (Li et al., 2026), SrcMarker (Yang et al., 2024), and RoSeMary (Zhang et al., 2025b). Token-level methods embed watermarks during generation, whereas post-hoc methods operate on generated code. ACW was evaluated only in Python because its implementation supports only that language. Appendix A provides details on code generation, baseline training, and functional evaluation.

Table 4: TPR@FPR5% under five code-editing attacks. AUROC results are in Table 9.

		Python						Java						C++					
Model	Method	None	Ref.	L-d	Cmt.	Ren.	LLM	None	Ref.	L-d	Cmt.	Ren.	LLM	None	Ref.	L-d	Cmt.	Ren.	LLM
Qwen3.5	KGW	91.43	79.29	92.14	17.86	87.86	34.29	81.63	73.47	78.91	63.27	73.47	40.82	86.00	84.00	86.00	54.00	82.67	42.67
	SWEET	86.43	82.86	87.86	12.14	85.71	31.43	79.59	80.27	79.59	24.49	75.51	25.85	86.67	84.00	86.67	40.00	80.67	33.33
	Unigram	86.43	76.43	86.43	20.00	92.86	31.43	82.31	81.63	80.95	46.94	85.71	42.86	70.00	70.00	70.00	31.33	72.67	29.33
	STONE	45.00	42.86	45.71	6.43	49.29	26.43	23.81	23.13	24.49	10.88	23.13	8.84	20.00	20.00	20.00	9.33	22.67	9.33
	STA-1	75.71	74.29	78.57	6.43	67.14	27.86	51.70	48.30	51.02	16.33	42.18	14.97	83.33	79.33	83.33	56.67	74.00	64.00
	SrcMarker	89.29	90.71	88.57	89.29	7.86	10.71	40.82	40.82	39.46	40.82	10.88	2.72	54.67	54.67	54.67	54.67	38.00	0.67
	RoSeMary	98.57	90.00	97.86	98.57	5.00	9.29	40.14	40.14	38.10	40.14	11.56	2.72	80.00	80.00	80.00	80.00	56.67	4.67
	ACW	89.29	0.00	93.57	88.57	3.57	0.00	–	–	–	–	–	–	–	–	–	–	–	–
	SEW (ours)	**100**	**99.29**	**100**	**100**	**100**	**54.29**	**96.60**	**95.92**	**95.24**	**96.60**	**96.60**	**75.51**	**98.00**	**90.00**	**98.00**	**98.00**	**98.00**	**65.33**
gemma-4	KGW	33.14	25.44	33.73	4.14	21.30	12.43	39.46	35.68	40.54	7.03	24.32	16.76	65.31	62.24	65.31	27.55	55.10	37.76
	SWEET	62.72	48.52	53.85	16.57	55.62	24.85	24.86	20.00	25.41	5.41	24.86	8.65	66.33	63.78	66.33	29.08	63.27	34.69
	Unigram	68.05	60.95	68.64	8.88	50.30	18.93	24.86	23.24	24.86	8.11	20.54	8.11	64.29	52.55	64.29	13.27	51.53	21.94
	STONE	33.14	34.32	31.36	8.28	18.93	8.88	12.97	13.51	12.97	4.32	4.32	5.41	23.47	32.65	23.47	10.71	11.22	8.67
	STA-1	7.10	5.92	7.10	1.18	21.30	5.33	3.24	3.24	3.24	2.70	2.16	0.54	9.18	6.63	9.18	1.02	7.65	2.04
	SrcMarker	90.53	89.35	87.57	90.53	7.10	9.47	86.49	86.49	85.95	86.49	34.59	5.41	89.29	89.29	89.29	89.29	58.67	6.63
	RoSeMary	97.04	86.39	95.27	97.04	5.92	5.92	25.41	25.41	24.86	25.41	9.19	1.62	90.82	90.82	90.82	90.82	56.12	4.59
	ACW	91.72	0.00	95.27	91.72	2.37	0.00	–	–	–	–	–	–	–	–	–	–	–	–
	SEW (ours)	**100**	**99.41**	**100**	**100**	**100**	**64.50**	**100**	**98.92**	**100**	**100**	**100**	**88.65**	**98.98**	**92.86**	**98.98**	**98.98**	**98.98**	**79.08**
gpt-oss	KGW	20.41	5.61	17.86	16.84	17.86	5.61	8.82	7.35	8.82	7.35	7.84	0.49	26.61	16.74	26.61	22.32	18.88	5.58
	SWEET	30.61	21.94	31.63	14.29	21.43	23.47	19.12	20.10	18.63	12.75	15.69	3.43	27.04	26.61	27.04	19.31	16.74	3.86
	Unigram	17.86	14.80	20.92	13.27	12.24	17.86	2.45	2.45	2.45	2.45	2.45	0.00	6.87	9.01	6.87	6.01	6.44	1.29
	STONE	6.12	1.02	6.12	5.10	4.59	2.55	6.37	5.88	5.88	6.37	6.86	2.45	27.47	21.03	27.47	29.18	34.33	23.61
	STA-1	22.96	18.88	23.47	17.35	15.82	9.18	7.84	6.37	8.82	7.35	11.76	6.37	41.63	30.47	41.63	37.34	40.34	40.77
	SrcMarker	90.82	90.82	90.31	90.82	9.18	8.67	87.25	87.25	86.27	87.25	30.39	7.35	65.67	65.67	65.67	65.67	41.20	9.01
	RoSeMary	97.96	89.80	97.45	97.96	4.59	4.08	27.45	27.45	27.45	27.45	8.33	0.49	72.96	72.96	72.96	72.96	42.92	4.29
	ACW	89.29	1.53	91.84	87.76	3.57	0.00	–	–	–	–	–	–	–	–	–	–	–	–
	SEW (ours)	**98.47**	**93.88**	**98.47**	**98.47**	**98.47**	**52.55**	**98.04**	**98.04**	**98.04**	**98.04**	**98.04**	**97.55**	**98.28**	**87.98**	**98.28**	**98.28**	**98.28**	**92.70**

5 Results and Analysis

5.1 Main Results

Table 3 reports detection and functional correctness across nine model–language settings. SEW achieved the highest TPR@FPR5% in all nine settings, ranging from 96.60% to 100%, with a mean of 98.71% and a mean relative improvement of 11.90% over the strongest baseline in each setting. It also achieved the highest AUROC in eight settings; in Qwen3.5 C++, KGW obtained 98.72 versus 98.43 for SEW. This small gap reflects coverage: SEW requires completed code that can be parsed and rewritten, whereas KGW embeds during generation and can retain detectable signals in incomplete or invalid outputs. Appendix G discusses limitations for short, incomplete, or structurally invalid code. SEW also preserved functional correctness, with pass@1 exactly matching Vanilla in all nine settings. Post-hoc failures showed that ACW could alter non-commutative string or list concatenation through operand reordering, while learned SrcMarker and RoSeMary could change loop behavior by replacing postfix with prefix decrement in loop conditions. SEW avoids these failures through rule-specific type, side-effect, and structural checks that restrict transformations to semantically valid sites. Token-level methods alter token selection during generation, so pass@1 can rise or fall relative to Vanilla without indicating consistent functional improvement. Overall, SEW preserved pass@1 while maintaining high detection performance.

5.2 Robustness to Code-Editing Attacks

Table 4 compares supported methods under five code-editing attacks: formatting (Ref.), default linting (L-d), comment removal (Cmt.), identifier renaming (Ren.), and LLM rewriting (LLM). Under the four non-LLM attacks, SEW achieved 97.78% mean TPR@FPR5%, with a 0.94% mean relative decrease. KGW and SWEET degraded substantially after comment removal, consistent with dependence on comment-related token evidence. SrcMarker and RoSeMary were less affected by comment removal but sensitive to identifier renaming. ACW degraded sharply under formatting that can overwrite its fixed style patterns. SEW applies code style rules only at eligible sites and draws watermark evidence from multiple code elements, reducing dependence on any single element. Comment removal and identifier renaming left its TPR@FPR5% unchanged in all nine settings, while formatting retained 87.98%–99.41%. LLM rewriting with Qwen3-Coder-30B-A3B-Instruct reduced SEW's TPR@FPR5% to 52.55%–97.55%, averaging 74.46%, but SEW remained

highest in all nine settings. SrcMarker and RoSeMary fell to 0.67%–10.71% and 0.49%–9.29%, respectively, while ACW reached 0% in all three Python settings. SEW embeds watermarks through context-appropriate code style choices, applying rules only where their semantic and structural conditions are satisfied. When these choices and their structural context survive rewriting, they continue to provide watermark evidence. This is consistent with SEW retaining the highest TPR@FPR5% across all nine settings under the evaluated LLM rewriting attack. Appendix D provides corresponding AUROC results and further analysis.

5.3 Robustness to Targeted Style Flipping

Figure 3 evaluates a targeted attack in which an attacker knows each method's watermark rules and reverses a fraction of its encoded style decisions at their corresponding sites. Unlike code-editing attacks, this attack targets the choices used to carry the watermark. SEW degraded gradually as more sites were reversed, retaining TPR@FPR5% of 81.52%–94.91% at 50% flipping and 52.32%–70.52% at 75% flipping across the three languages. At 75%, SrcMarker achieved 27.50%–39.34% and RoSeMary 18.05%–42.16%. ACW was substantially more sensitive: its TPR@FPR5% dropped from 90.10% without attack to 40.57% at 5% flipping and 20.49% at 10%. SEW groups repeated sites that share the same structural identifier and context into a grade and represents them by a single majority observation. Thus, when only part of the sites within a grade are flipped, its watermark evidence can remain unchanged until the majority is reversed. Complete flipping drives SEW to 0% because every encoded site is forced to the opposite target. SrcMarker and RoSeMary retain nonzero detection even under complete style flipping. This residual detection comes from identifier-based watermark components that are not modified by the attack. Overall, SEW remains detectable even when an attacker directly reverses a large fraction of its encoded style decisions.

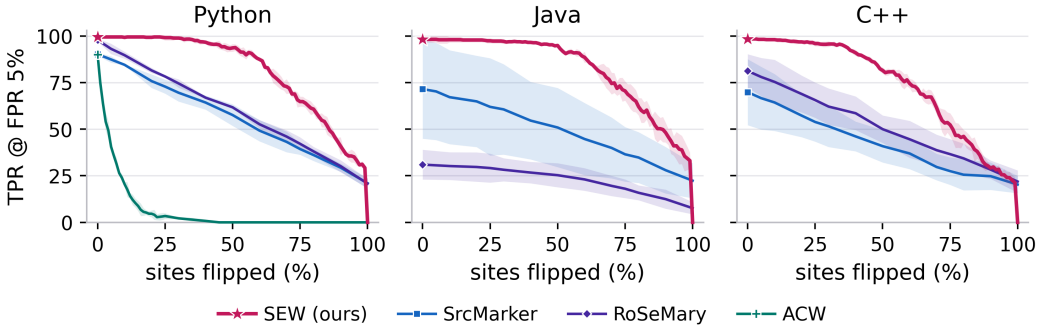

Figure 3: TPR@FPR5% under targeted style flipping; lines show the three-backbone mean and shaded bands the standard deviation.

5.4 Resistance to Rule Inference

Table 5 evaluates whether an attacker can predict watermark decisions after observing N watermarked programs. Knowing the rules, the attacker uses observed decisions to predict style variants on unseen test problems. Predictions must match the watermarked test programs; unseen decisions count as incorrect. We report mean accuracy over ten runs per N; lower values indicate poorer transfer. We report Python, the only language supported by all four methods. Appendix E provides the results for all model–language settings. The baselines were highly predictable at $N = 10$, reaching 85.97%–99.67% accuracy, whereas SEW achieved 19.91%. At $N = 100$, SEW reached 45.17%, while baselines remained above 85%. This contrast reflects decision assignment: ACW applies each rule in a fixed direction, so observed decisions transfer directly to new programs. SrcMarker and RoSeMary use learned embedding, but their rule-level transformations recur consistently and remain predictable across programs. SEW instead derives each target from $\mathrm{siteID}(s)$ and program context $\chi(C)$. Thus, knowing a rule and its decisions elsewhere does not reveal its target in a new program; the corresponding siteID–$\chi(C)$ combination must already have been observed. As N increases, more combinations become known, but unseen ones still limit prediction when the same rule recurs.

Table 5: Python rule-inference accuracy (%) over ten runs, averaged across three backbones; subscripts denote standard deviations across the backbones.

Method	N=10	N=30	N=50	N=70	N=90	N=100
SrcMarker	$90.15_{\pm 0.72}$	$90.52_{\pm 0.87}$	$90.37_{\pm 0.69}$	$90.53_{\pm 0.70}$	$90.98_{\pm 0.65}$	$90.54_{\pm 0.43}$
RoSeMary	$85.97_{\pm 0.35}$	$85.65_{\pm 0.34}$	$86.16_{\pm 0.35}$	$85.69_{\pm 0.12}$	$85.96_{\pm 0.13}$	$85.87_{\pm 0.31}$
ACW	$99.67_{\pm 0.12}$	$99.82_{\pm 0.03}$	$99.90_{\pm 0.02}$	$99.90_{\pm 0.03}$	$99.92_{\pm 0.01}$	$99.91_{\pm 0.04}$
SEW (ours)	$\mathbf{19.91}_{\pm 6.42}$	$\mathbf{35.83}_{\pm 7.70}$	$\mathbf{39.79}_{\pm 5.83}$	$\mathbf{42.01}_{\pm 7.14}$	$\mathbf{45.63}_{\pm 3.17}$	$\mathbf{45.17}_{\pm 7.10}$

5.5 Ablation Study

Table 6 isolates the effects of style-preference calibration and grade-level detection. Both components improve T@5 without attacks. For gemma-4 on Java, calibration alone raises T@5 from 89.19% to 97.84%, while grade-level detection reaches 100%. For gpt-oss on C++, the corresponding values are 88.41%, 97.85%, and 98.28%. The combined configuration improves over Base in seven of the nine model–language settings; the remaining two already achieve 100%. However, its T@5 matches grade-level detection alone in all nine settings, indicating that adding calibration after aggregation provides no further improvement at this operating point. The benefits of combining components become clearer under attacks. The ranges summarize language-specific averages over three models. Calibration alone raises Atk. from 72.54%–73.51% to 81.66%–90.62% and Flip from 30.77%–34.49% to 58.63%–70.46%. Grade-level detection gives higher Atk. of 95.54%–97.69%, but neither component consistently dominates under Flip: grade-level detection is stronger in Python and C++, whereas calibration is stronger in Java. Combining both achieves the highest Atk. and Flip in every language, reaching 96.38%–99.00% and 76.59%–87.63%, respectively. Calibration accounts for natural style frequencies, while grade-level detection aggregates repeated observations and includes the complementary syntax-only test. Their combined gains under attacks, despite unchanged T@5 relative to grade-level detection alone, show that the components provide complementary benefits when watermark evidence is modified.

Table 6: Ablating calibration and grade-level detection. T@5, Atk., and Flip report TPR@FPR5% without attack, under four non-LLM editing attacks, and under flipping attacks.

				Python			**Java**			**C++**		
Model	Configuration	Calib	Grade	T@5	Atk.	Flip	T@5	Atk.	Flip	T@5	Atk.	Flip
Qwen3.5	Base	✗	✗	**100**	74.82	36.67	95.92	77.55	38.55	94.00	73.83	34.22
		✓	✗	**100**	86.78	71.19	95.92	90.14	73.02	96.67	81.00	63.78
		✗	✓	**100**	99.11	79.52	**96.60**	95.41	68.03	**98.00**	94.50	66.45
	SEW (ours)	✓	✓	**100**	**99.82**	**86.91**	**96.60**	**96.09**	**88.89**	**98.00**	**96.00**	**76.89**
gemma-4	Base	✗	✗	**100**	76.18	38.07	89.19	70.54	29.01	95.41	75.13	31.63
		✓	✗	**100**	88.02	71.20	97.84	90.41	67.93	98.47	83.29	57.31
		✗	✓	**100**	99.56	77.32	**100**	97.43	65.95	**98.98**	97.07	68.20
	SEW (ours)	✓	✓	**100**	**99.85**	**87.18**	**100**	**99.73**	**87.39**	**98.98**	**97.45**	**74.49**
gpt-oss	Base	✗	✗	96.94	69.52	28.74	89.71	71.57	35.62	88.41	68.67	26.47
		✓	✗	**98.47**	78.83	68.71	**98.04**	91.30	70.43	97.85	80.68	54.79
		✗	✓	**98.47**	94.39	72.62	**98.04**	96.69	67.97	**98.28**	95.06	69.53
	SEW (ours)	✓	✓	**98.47**	**97.32**	**83.50**	**98.04**	**98.04**	**86.60**	**98.28**	**95.70**	**78.40**

6 Conclusion

SEW watermarks completed LLM-generated code without model training or decoding access. It selects code styles using a secret key and structural context, combining rule-specific applicability checks, style-preference calibration, and context-aware style aggregation. Across three LLMs and three languages on CodeContests, SEW achieved 98.71% mean TPR@FPR5% and an 11.90% mean relative improvement over each setting's strongest baseline, preserving Vanilla pass@1. Four non-LLM code-editing attacks yielded a 0.94% mean relative TPR@FPR5% decrease. SEW's code style-based post-hoc watermarking combines robust detection, functional preservation, and resistance to rule inference from multiple watermarked programs.

AI USE STATEMENT

We used ChatGPT to correct grammatical errors and refine word choice in the manuscript. Separately, the LLMs listed in Section 4.2 generated code for the experiments. SEW performs post-hoc embedding and detection without LLM calls. The authors take responsibility for the research methodology, experimental results, references, and final manuscript, including content prepared with AI assistance.

ETHICS STATEMENT

SEW is intended to provide statistical evidence of a key-dependent watermark rather than determine whether arbitrary code was written by a human or an LLM. A positive detection does not establish the identity of the author or ownership of the code, and a negative detection does not establish the absence of LLM assistance. False positives can lead to incorrect attribution, while code transformations can remove watermark evidence. Detection results should therefore be considered alongside independent evidence and should not serve as the sole basis for allegations of misconduct. Our evaluation of functional correctness is limited to the CodeContests test cases. Passing these tests does not establish semantic equivalence for every input or the absence of security vulnerabilities. Watermark detection and benchmark correctness should therefore not be interpreted as guarantees of software safety.

REPRODUCIBILITY STATEMENT

Section 4 specifies the evaluation datasets and models, together with the metrics used to assess watermark detection and functional correctness. Appendix A describes the experimental hardware, code-generation settings, baseline training data and procedures, and functional evaluation using the CodeContests test cases. It also specifies the source and size of the separate corpus used for style-preference calibration. Appendix B details problem selection and truncation filtering, including retained counts for each model–language setting and the treatment of embedding failures during evaluation. Appendix C reports calibration corpus sizes and the time required to estimate natural style probabilities for each language. Appendix F provides the full rule list, binary style variants, and supported languages, with examples of rule-specific type, side-effect, and structural applicability conditions. The abstract provides a link to the anonymous code repository.

REFERENCES

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, pp. 281–293, 2014.

Georg Brandl and Serhiy Storchaka. PEP 515 – underscores in numeric literals. `https://peps.python.org/pep-0515/`, 2016.

Aylin Caliskan-Islam, Richard E. Harang, Andrew Liu, Arvind Narayanan, Clare R. Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Security Symposium, USENIX Security*, pp. 255–270, 2015.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, pp. 18–30, 2022.

Yitong Chen, Zhiqiang Gao, Chuanqi Shi, Baixuan Li, and Miao Gao. ABC: towards a universal code styler through model merging. *Proceedings of the ACM on Programming Languages*, 9 (OOPSLA2):1512–1540, 2025.

Zhenlong Dai, Chang Yao, WenKang Han, Ying Yuan, Zhipeng Gao, and Jingyuan Chen. MPCoder: Multi-user personalized code generator with explicit and implicit style representation learning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 3765–3780, 2024.

Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, Mayank Chaturvedi, Aditya Chawla, Victor Cotruta, Alice Coucke, Phil Culliton, Robert Dadashi, Lucas Dixon, Mohamed Elhawaty, Utku Evci, Clément Farabet, Johan Ferret, Filippo Galgani, Sertan Girgin, Jean-Bastien Grill, Maarten Grootendorst, Jiaxian Guo, Cassidy Hardin, Yanzhang He, Steven M. Hernandez, Omri Homburger, Léonard Hussenot, Ju-yeong Ji, Armand Joulin, Aishwarya Kamath, Parnian Kassraie, Olivier Lacombe, Preethi Lahoti, Gaël Liu, Gus Martins, Luciano Martins, Tatiana Matejovicova, Ramona Merhej, Nikola Momchev, Sneha Mondal, Ryan Mullins, Sindhu Raghuram Panyam, Shreya Pathak, Sarah Perrin, André Susano Pinto, Etienne Pot, Angéline Pouget, Alexandre Ramé, Sabela Ramos, Douglas Reid, David Rim, Morgane Rivière, Karsten Roth, Louis Rouillard, Omar Sanseviero, Pier Giuseppe Sessa, Shane Settle, Danila Sinopalnikov, Sara Smoot, Piotr Stanczyk, Andreas Steiner, Lawrence Stewart, Ilya O. Tolstikhin, Michael Tschannen, Anton Tsitsulin, Nino Vieillard, Renjie Wu, Pingmei Xu, Haichuan Yang, Edouard Yvinec, Biao Zhang, Li Zhang, Joe Zou, Nicolas Aagnes, Abdelrahman Abdelhamed, Jakub Adámek, Shivani Agrawal, Shubham Agrawal, Ibrahim Alabdulmohsin, Jean-Baptiste Alayrac, Uri Alon, Chandramouli Amarnath, Ankesh Anand, Chrysovalantis Anastasiou, Setareh Ariafar, François-Xavier Aubet, Kyriakos Axiotis, Federico Barbero, Joelle K. Barral, Alexei Bendebury, Urs Bergmann, Stanley Bileschi, Kat Black, Mathieu Blondel, Sebastian Borgeaud, Arthur Bražinskas, et al. Gemma 4 technical report. arXiv preprint arXiv:2607.02770, 2026.

Google. Google style guides. `https://google.github.io/styleguide/`, n.d.

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

kaysss. LeetCode solution dataset. `https://huggingface.co/datasets/kaysss/leetcode-problem-solutions`, 2025.

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*, pp. 3990–4002, 2026.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. A watermark for large language models. In *Proceedings of the 40th International Conference on Machine Learning, ICML*, volume 202, pp. 17061–17084, 2023.

Łukasz Langa and contributors to Black. Black: The uncompromising Python code formatter. `https://github.com/psf/black`, 2026.

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, ACL*, pp. 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. *IEEE Transactions on Software Engineering*, 2026.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022a.

Zhen Li, Qian (Guenevere) Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. RoPGen: Towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering, ICSE*, pp. 1906–1918, 2022b.

Minjia Mao, Dongjun Wei, Zeyu Chen, Xiao Fang, and Michael Chau. Watermarking large language models: An unbiased and low-risk method. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 7939–7960, 2025.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. arXiv preprint arXiv:2508.10925, 2025.

Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading authorship attribution of source code using adversarial learning. In *Proceedings of the 28th USENIX Security Symposium, USENIX Security*, pp. 479–496, 2019.

Qwen Team. Qwen3.5: Towards native multimodal agents. `https://qwen.ai/blog?id=qwen3.5`, 2026.

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, ESEC/FSE*, pp. 1561–1572, 2023.

Sun Microsystems. Code conventions for the Java programming language. `https://www.oracle.com/java/technologies/javase/codeconventions-contents.html`, 1999.

Chih-Kai Ting, Karl Munson, Serenity Wade, Anish Savla, Kiran Kate, and Kavitha Srinivas. CodeStylist: A system for performing code style transfer using neural networks. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI*, pp. 16485–16487, 2023.

Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. PEP 8 – style guide for Python code. `https://peps.python.org/pep-0008/`, 2001.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 13818–13843, 2023.

Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. Beyond functional correctness: Investigating coding style inconsistencies in large language models. *Proceedings of the ACM on Software Engineering*, 2(FSE):690–712, 2025.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP*, pp. 38–45, 2020.

Borui Yang, Wei Li, Liyao Xiang, and Bo Li. SrcMarker: Dual-channel source code watermarking via scalable code transformations. In *Proceedings of the 45th IEEE Symposium on Security and Privacy, SP*, pp. 4088–4106, 2024.

Dutao Zhang, Sergey V. Kovalchuk, and Yulong He. Style2Code: A style-controllable code generation framework with dual-modal contrastive representation learning. arXiv preprint arXiv:2505.19442, 2025a.

Ruisi Zhang, Neusha Javidnia, Nojan Sheybani, and Farinaz Koushanfar. Robust and secure code watermarking for large language models via ML/crypto codesign. arXiv preprint arXiv:2502.02068, 2025b.

Xuandong Zhao, Prabhanjan Vijendra Ananth, Lei Li, and Yu-Xiang Wang. Provable robust watermarking for AI-generated text. In *Proceedings of the 12th International Conference on Learning Representations, ICLR*, 2024.

A Implementation Details

We conducted the experiments using three NVIDIA RTX PRO 6000 GPUs. We generated unwatermarked outputs, denoted as Vanilla, using Hugging Face Transformers (Wolf et al., 2020) with greedy decoding and a maximum of 8,192 new tokens. SEW and the post-hoc baselines were applied to these same outputs. Token-level baselines used the configurations recommended in their respective papers and implementations. Functional correctness was evaluated using pass@1 on all test cases in the public_tests, private_tests, and generated_tests fields of each CodeContests problem. A program was considered correct only if it passed all these tests.

For style-preference calibration, we used user-posted LeetCode solutions. The calibration corpus contains 11,834 Python, 10,643 Java, and 11,594 C++ files. We estimated natural style probabilities from this corpus without using the CodeContests evaluation solutions. We trained all language-specific SrcMarker models for 25 epochs using the authors' implementation. Our RoSeMary reimplementation was trained for 20 epochs on the same datasets. For both methods, the Python and Java models each used 157,851 training functions from the corresponding filtered CodeSearchNet (Husain et al., 2019) dataset. The models evaluated on C++ were trained on 3,661 C functions from GitHub-C. Neither method was trained on CodeContests.

B Data Construction

We used the validation and test splits of CodeContests (Li et al., 2022a), retaining problems with a canonical human solution in the target language. The initial pools contained 203 Python, 223 Java, and 267 C++ problems. For each model–language setting, we generated unwatermarked outputs (Vanilla) with a maximum of 8,192 new tokens. We excluded problems whose Vanilla outputs exhausted this budget without completing a code block, including outputs with an unclosed code fence and those that never began a code block. This filtering focuses the comparison on watermarking completed code. Token-level methods can retain detectable signals even in reasoning-only outputs, whereas post-hoc methods require generated code as input; detecting signals without generated code falls outside this evaluation scope. The same Vanilla-based filtering criterion and retained problem set were used for all methods within each setting, including token-level methods that generated watermarked code separately. Table 7 reports the resulting problem counts. Filtering did not require compilation or test success, and embedding failures and abstentions within the retained sets remained in the detection denominator with a score of zero. Thus, the filtering did not select programs based on whether a particular watermarking method succeeded.

Inspection of all truncated outputs identified two distinct patterns. For Qwen3.5-9B and gemma-4-12B-it, reasoning continued within code comments. Comments accounted for 91.7%, 89.6%, and 91.3% of lines in truncated Qwen3.5-9B outputs for Python, Java, and C++, respectively. The corresponding proportions for gemma-4-12B-it were 91.8%, 89.0%, and 93.1%. Among the 63 truncated Qwen3.5-9B Python outputs, 61 contained at least 60% comment lines. In contrast, gpt-oss-20b exhausted the token budget during reasoning before opening a code block, leaving little or no generated code. The excluded outputs therefore included cases where reasoning within code comments exhausted the allotted token budget before generation finished, as well as cases where reasoning exhausted the budget before code generation began.

Table 7: CodeContests problem counts before and after model-specific truncation filtering.

Problem set	Python	Java	C++
Before filtering	203	223	267
Qwen3.5-9B	140	147	150
gemma-4-12B-it	169	185	196
gpt-oss-20b	196	204	233

C STYLE-PREFERENCE CALIBRATION COST

SEW estimates natural variant probabilities by counting eligible occurrences of each style variant, without neural model training. The procedure applies the fixed rule library and applicability checks to the calibration corpus and converts the observed counts into rule-specific probabilities. The rules are not learned or modified during calibration. As shown in Table 8, estimation took 23.2–24.7 seconds per language on corpora containing 10,643–11,834 LeetCode files. The probability tables are reused during detection, so calibration is not repeated for each inspected program.

Table 8: Size of the LeetCode calibration corpus and the time taken to estimate the rule-specific natural variant probabilities for each language.

Language	Files	Time (s)
Python	11,834	24.1
Java	10,643	23.2
C++	11,594	24.7

D AUROC ANALYSIS UNDER CODE-EDITING ATTACKS

Table 9 complements the TPR@FPR5% results with AUROC under five code-editing attacks: formatting, default linting, comment removal, identifier renaming, and LLM rewriting. All AUROC values are scaled by 100. Under the four non-LLM attacks, SEW achieved 96.30–100 across the 36 model–language–attack settings. Identifier renaming left its AUROC unchanged in all nine settings, while comment removal changed it by at most 0.01. This stability is consistent with SEW identifying sites and assigning targets through structural context rather than identifier names or comment content. Formatting had a larger effect, but SEW retained AUROC of 96.60–99.76.

The baselines exhibit attack-specific weaknesses. For Qwen3.5 on Python, comment removal reduced KGW's AUROC from 97.24 to 71.30 and SWEET's from 94.39 to 59.61, consistent with dependence on comment-related token evidence. SrcMarker and RoSeMary retained their AUROC after comment removal, but identifier renaming reduced RoSeMary's Python AUROC to 48.90–49.62 across the three models. ACW similarly fell to 50.00–50.77 under formatting, consistent with a formatter overwriting its fixed style patterns. These results show that resistance to an edit affecting one code element does not imply resistance to edits affecting other elements. SEW retained high AUROC across all four categories in all three programming languages rather than depending on the preservation of comments, names, or formatting alone.

LLM rewriting exposed a larger overall loss in detection performance. SEW retained AUROC of 80.88–98.44, exceeding all supported post-hoc baselines in every model–language setting and achieving the highest AUROC among all methods in seven of the nine settings. SrcMarker reached 43.46–79.94, RoSeMary reached 45.27–53.88, and ACW reached 50.00 in all three Python settings. The contrast with the preceding attacks shows that robustness to individual code edits does not fully extend to LLM rewriting. For SEW, rewriting that changes $\mathrm{siteID}(s)$ or the program context $\chi(C)$ can change the recomputed target even when a local style variant remains unchanged. Thus, preserving surface-level choices alone does not ensure that the detector reconstructs the same watermark evidence. The results demonstrate partial robustness to the evaluated rewriting attack, rather than invariance to arbitrary structural changes.

AUROC and TPR@FPR5% also distinguish overall score separation from detection at a low false-positive rate. For gpt-oss on Python, LLM rewriting reduced SEW's AUROC from 98.91 to 89.63, while TPR@FPR5% fell from 98.47% to 52.55%. A relatively high AUROC therefore does not guarantee high TPR@FPR5%. Conversely, under LLM rewriting of Qwen3.5 C++ outputs, STA-1 achieved higher AUROC than SEW (90.43 versus 80.88), but lower TPR@FPR5% (64.00% versus 65.33%). Both metrics are therefore needed: AUROC characterizes ranking across thresholds, while TPR@FPR5% identifies how much detection remains at the evaluated operating point.

Table 9: AUROC under five code-editing attacks.

		Python						Java						C++					
Model	Method	None	Ref.	L-d	Cmt.	Ren.	LLM	None	Ref.	L-d	Cmt.	Ren.	LLM	None	Ref.	L-d	Cmt.	Ren.	LLM
Qwen3.5	KGW	97.24	86.41	97.30	71.30	95.08	70.02	96.67	93.69	96.11	93.20	93.19	**85.92**	**98.72**	**97.24**	**98.72**	93.65	98.28	86.97
	SWEET	94.39	91.42	96.38	59.61	92.99	72.20	91.12	91.37	91.30	69.43	84.96	59.57	94.29	92.94	94.29	72.12	90.08	66.26
	Unigram	96.05	86.25	95.91	65.48	97.94	74.12	93.65	93.52	92.98	81.54	96.47	82.99	85.51	85.21	85.51	58.82	89.30	53.84
	STONE	67.33	65.17	68.04	57.10	73.16	65.75	66.85	63.73	67.88	60.15	66.52	56.27	43.33	39.73	43.33	42.92	49.83	40.93
	STA-1	93.89	89.78	94.14	67.38	85.04	73.91	85.41	83.27	86.39	66.54	75.78	67.61	96.05	94.85	96.05	88.58	91.59	**90.43**
	SrcMarker	98.75	98.82	98.71	98.75	80.35	79.94	91.95	91.95	90.26	91.95	56.92	43.63	79.72	79.72	79.72	79.72	73.71	46.74
	RoSeMary	98.21	97.45	98.13	98.21	48.90	49.05	94.70	94.70	92.24	94.70	66.48	45.27	89.32	89.32	89.32	89.32	84.44	52.39
	ACW	94.64	50.00	96.79	94.29	51.79	50.00	–	–	–	–	–	–	–	–	–	–	–	–
	SEW (ours)	**100**	**99.37**	**99.86**	**100**	**100**	**83.60**	**97.37**	**96.82**	**96.30**	**97.37**	**97.37**	84.46	98.43	96.60	98.43	**98.43**	**98.43**	80.88
gemma-4	KGW	74.29	64.84	74.15	49.87	59.95	53.24	77.38	75.02	78.54	58.05	62.87	55.69	90.95	89.18	90.95	77.21	86.32	79.01
	SWEET	88.16	81.41	84.48	73.03	86.21	68.77	71.98	61.70	74.44	49.76	73.15	60.39	91.99	90.27	91.99	77.74	90.19	78.03
	Unigram	95.84	91.94	95.79	76.49	90.29	74.08	71.11	63.85	71.01	46.30	53.74	53.58	92.84	89.46	92.84	74.89	88.65	70.25
	STONE	64.47	66.51	64.44	46.15	52.48	37.69	65.77	65.17	61.75	55.80	36.64	54.01	71.17	78.88	71.17	65.83	49.11	56.14
	STA-1	33.89	33.45	34.04	29.24	69.93	46.10	51.61	47.12	55.41	55.61	40.30	43.98	36.92	31.88	36.92	19.74	31.89	17.44
	SrcMarker	97.64	97.63	97.11	97.64	76.03	75.99	94.39	94.39	94.36	94.39	63.63	47.22	91.96	91.96	91.96	91.96	84.76	54.76
	RoSeMary	97.07	96.50	96.98	97.07	49.32	53.88	95.47	95.47	95.26	95.47	66.05	46.97	95.06	95.06	95.06	95.06	87.30	50.71
	ACW	95.86	50.00	97.63	95.86	51.18	50.00	–	–	–	–	–	–	–	–	–	–	–	–
	SEW (ours)	**100**	**99.40**	**99.96**	**100**	**100**	**88.90**	**100**	**99.76**	**100**	**100**	**100**	**93.85**	**99.20**	**98.08**	**99.20**	**99.19**	**99.20**	**90.76**
gpt-oss	KGW	70.14	47.56	68.10	67.12	65.56	50.00	42.87	38.16	45.05	37.91	35.05	25.16	63.94	44.92	63.94	59.54	45.13	43.30
	SWEET	72.98	59.63	71.56	63.31	60.49	72.60	66.57	63.27	65.45	59.08	49.42	50.26	75.44	73.16	75.44	65.52	44.44	44.77
	Unigram	75.02	70.51	76.47	69.69	64.67	73.33	49.34	50.99	48.52	48.18	34.36	40.19	35.03	49.16	35.03	30.64	22.39	19.13
	STONE	55.19	41.03	55.33	52.06	37.88	37.25	56.24	54.63	50.11	55.73	43.38	59.63	79.87	72.40	79.87	78.55	82.37	74.10
	STA-1	70.99	65.52	72.53	66.59	63.98	60.71	47.43	42.39	51.02	44.10	61.45	52.89	88.19	84.52	88.19	86.05	89.52	85.78
	SrcMarker	97.18	97.11	97.01	97.18	73.25	74.22	94.40	94.40	93.90	94.40	62.28	43.46	72.98	72.98	72.98	72.98	65.05	45.64
	RoSeMary	96.68	96.00	96.44	96.68	49.62	53.28	96.22	96.22	96.15	96.22	66.56	48.21	83.94	83.94	83.94	83.94	75.26	51.42
	ACW	94.64	50.77	95.92	93.88	51.79	50.00	–	–	–	–	–	–	–	–	–	–	–	–
	SEW (ours)	**98.91**	**97.74**	**98.86**	**98.91**	**98.91**	**89.63**	**98.51**	**98.50**	**98.51**	**98.51**	**98.51**	**98.44**	**98.65**	**96.68**	**98.65**	**98.65**	**98.65**	**97.49**

E Resistance to Rule Inference

Tables 10, 11, and 12 cover nine model–language settings. Knowing the rules, the attacker predicted styles in unseen test programs from $N \in \{10, 30, 50, 70, 90, 100\}$ observed watermarked programs. Predictions had to match watermarked test-code variants; unobserved decisions counted as errors. We report ten-run mean accuracy per N, with subscripted standard deviations; lower is better. Python, Java, and C++ test counts were 40, 41, and 19 for Qwen3.5; 69, 84, and 82 for gemma-4; and 93, 100, and 71 for gpt-oss. Methods shared test problems per setting; ACW supports only Python. SEW achieved the lowest mean accuracy throughout. At $N = 10$, baselines reached 85.70%–99.77%, versus 3.89%–26.85% for SEW. At $N = 100$, baselines remained at 85.53%–99.95%, while SEW reached 39.51%–53.14% in Python, 22.78%–26.86% in Java, and 17.53%–32.93% in C++. ACW reuses fixed directions; SrcMarker and RoSeMary showed recurring rule-level preferences despite learned embedding and random messages. SEW targets depend on the secret key, $\mathrm{siteID}(s)$, and program context $\chi(C)$. Under a fixed key, targets transfer to matching site–context combinations, but not necessarily across contexts for the same rule. Accuracy generally rose with N, consistent with more reusable combinations; this attack measures decision transfer, not secret-key recovery or unrestricted target prediction.

Table 10: Rule-inference accuracy (%) on Qwen3.5-9B.

Language	Method	N=10	N=30	N=50	N=70	N=90	N=100
Python	SrcMarker	$90.12_{\pm 1.19}$	$90.73_{\pm 1.53}$	$90.55_{\pm 2.04}$	$90.90_{\pm 1.40}$	$91.04_{\pm 1.70}$	$90.71_{\pm 2.16}$
	RoSeMary	$85.85_{\pm 1.11}$	$85.34_{\pm 1.64}$	$86.54_{\pm 1.32}$	$85.79_{\pm 1.25}$	$85.84_{\pm 0.86}$	$85.53_{\pm 0.84}$
	ACW	$99.54_{\pm 0.30}$	$99.84_{\pm 0.15}$	$99.92_{\pm 0.08}$	$99.90_{\pm 0.08}$	$99.92_{\pm 0.09}$	$99.89_{\pm 0.06}$
	SEW (ours)	$\mathbf{14.18}_{\pm 5.92}$	$\mathbf{29.06}_{\pm 5.56}$	$\mathbf{35.90}_{\pm 5.87}$	$\mathbf{36.39}_{\pm 5.53}$	$\mathbf{44.89}_{\pm 9.77}$	$\mathbf{39.51}_{\pm 9.30}$
Java	SrcMarker	$94.16_{\pm 0.86}$	$94.49_{\pm 0.35}$	$94.64_{\pm 0.49}$	$94.54_{\pm 0.42}$	$94.42_{\pm 0.43}$	$94.72_{\pm 0.41}$
	RoSeMary	$94.49_{\pm 1.16}$	$95.16_{\pm 0.46}$	$95.24_{\pm 0.66}$	$95.08_{\pm 0.72}$	$95.08_{\pm 0.32}$	$94.97_{\pm 0.42}$
	SEW (ours)	$\mathbf{6.13}_{\pm 3.88}$	$\mathbf{15.38}_{\pm 2.88}$	$\mathbf{17.20}_{\pm 2.81}$	$\mathbf{17.95}_{\pm 3.86}$	$\mathbf{18.97}_{\pm 3.69}$	$\mathbf{22.78}_{\pm 4.47}$
C++	SrcMarker	$93.28_{\pm 0.92}$	$93.80_{\pm 1.14}$	$93.58_{\pm 1.09}$	$93.91_{\pm 0.61}$	$94.38_{\pm 0.76}$	$93.56_{\pm 0.51}$
	RoSeMary	$94.71_{\pm 0.59}$	$94.64_{\pm 1.07}$	$94.94_{\pm 0.87}$	$94.97_{\pm 0.57}$	$94.54_{\pm 0.77}$	$94.85_{\pm 0.91}$
	SEW (ours)	$\mathbf{3.89}_{\pm 3.14}$	$\mathbf{7.62}_{\pm 5.05}$	$\mathbf{10.61}_{\pm 3.90}$	$\mathbf{16.87}_{\pm 6.09}$	$\mathbf{19.73}_{\pm 6.46}$	$\mathbf{21.64}_{\pm 5.21}$

Table 11: Rule-inference accuracy (%) on gemma-4-12B-it.

Language	Method	N=10	N=30	N=50	N=70	N=90	N=100
Python	SrcMarker	$89.45_{\pm 1.26}$	$89.57_{\pm 1.07}$	$89.61_{\pm 1.14}$	$89.72_{\pm 1.24}$	$90.30_{\pm 0.66}$	$90.06_{\pm 0.88}$
	RoSeMary	$85.70_{\pm 0.89}$	$85.58_{\pm 0.75}$	$85.85_{\pm 1.44}$	$85.56_{\pm 0.86}$	$85.94_{\pm 0.62}$	$86.14_{\pm 0.64}$
	ACW	$99.77_{\pm 0.12}$	$99.79_{\pm 0.10}$	$99.88_{\pm 0.08}$	$99.93_{\pm 0.04}$	$99.91_{\pm 0.06}$	$99.88_{\pm 0.07}$
	SEW (ours)	$\mathbf{26.85}_{\pm 7.16}$	$\mathbf{44.20}_{\pm 4.24}$	$\mathbf{46.49}_{\pm 3.10}$	$\mathbf{50.04}_{\pm 4.22}$	$\mathbf{49.10}_{\pm 2.27}$	$\mathbf{53.14}_{\pm 4.31}$
Java	SrcMarker	$94.33_{\pm 0.78}$	$94.39_{\pm 0.91}$	$94.71_{\pm 0.33}$	$94.66_{\pm 0.39}$	$94.83_{\pm 0.34}$	$94.62_{\pm 0.28}$
	RoSeMary	$94.31_{\pm 1.05}$	$94.68_{\pm 1.03}$	$95.08_{\pm 0.20}$	$95.18_{\pm 0.34}$	$94.93_{\pm 0.36}$	$95.10_{\pm 0.44}$
	SEW (ours)	$\mathbf{5.72}_{\pm 2.28}$	$\mathbf{11.37}_{\pm 2.44}$	$\mathbf{15.81}_{\pm 2.69}$	$\mathbf{20.40}_{\pm 2.75}$	$\mathbf{24.50}_{\pm 2.90}$	$\mathbf{24.08}_{\pm 3.71}$
C++	SrcMarker	$93.26_{\pm 0.59}$	$93.55_{\pm 0.38}$	$93.17_{\pm 0.61}$	$93.49_{\pm 0.35}$	$93.62_{\pm 0.23}$	$93.57_{\pm 0.22}$
	RoSeMary	$94.29_{\pm 1.09}$	$94.57_{\pm 0.79}$	$95.00_{\pm 0.29}$	$94.84_{\pm 0.33}$	$94.85_{\pm 0.34}$	$94.82_{\pm 0.31}$
	SEW (ours)	$\mathbf{4.66}_{\pm 1.86}$	$\mathbf{9.23}_{\pm 1.48}$	$\mathbf{12.54}_{\pm 2.27}$	$\mathbf{15.76}_{\pm 2.54}$	$\mathbf{18.82}_{\pm 2.57}$	$\mathbf{17.53}_{\pm 1.62}$

Table 12: Rule-inference accuracy (%) on gpt-oss-20b.

Language	Method	N=10	N=30	N=50	N=70	N=90	N=100
Python	SrcMarker	$90.88_{\pm 1.57}$	$91.27_{\pm 1.37}$	$90.96_{\pm 1.65}$	$90.96_{\pm 1.27}$	$91.60_{\pm 0.86}$	$90.86_{\pm 1.58}$
	RoSeMary	$86.36_{\pm 0.87}$	$86.02_{\pm 0.66}$	$86.08_{\pm 0.64}$	$85.71_{\pm 0.83}$	$86.09_{\pm 0.80}$	$85.95_{\pm 0.64}$
	ACW	$99.70_{\pm 0.21}$	$99.83_{\pm 0.13}$	$99.90_{\pm 0.07}$	$99.88_{\pm 0.06}$	$99.93_{\pm 0.04}$	$99.95_{\pm 0.03}$
	SEW (ours)	$\mathbf{18.69}_{\pm 6.08}$	$\mathbf{34.23}_{\pm 4.01}$	$\mathbf{36.98}_{\pm 4.19}$	$\mathbf{39.61}_{\pm 2.63}$	$\mathbf{42.89}_{\pm 3.83}$	$\mathbf{42.85}_{\pm 4.51}$
Java	SrcMarker	$94.32_{\pm 0.89}$	$94.86_{\pm 0.27}$	$94.69_{\pm 0.24}$	$94.72_{\pm 0.21}$	$94.66_{\pm 0.28}$	$94.72_{\pm 0.29}$
	RoSeMary	$94.68_{\pm 0.99}$	$94.99_{\pm 0.31}$	$95.31_{\pm 0.33}$	$95.10_{\pm 0.41}$	$95.11_{\pm 0.36}$	$95.09_{\pm 0.34}$
	SEW (ours)	$\mathbf{5.40}_{\pm 2.38}$	$\mathbf{12.99}_{\pm 2.88}$	$\mathbf{17.87}_{\pm 3.24}$	$\mathbf{21.42}_{\pm 3.79}$	$\mathbf{28.04}_{\pm 1.39}$	$\mathbf{26.86}_{\pm 4.56}$
C++	SrcMarker	$93.54_{\pm 0.50}$	$93.38_{\pm 0.81}$	$93.54_{\pm 0.61}$	$93.81_{\pm 0.69}$	$93.53_{\pm 0.27}$	$93.76_{\pm 0.23}$
	RoSeMary	$94.43_{\pm 1.07}$	$95.09_{\pm 0.21}$	$94.94_{\pm 0.36}$	$95.10_{\pm 0.37}$	$95.09_{\pm 0.28}$	$94.99_{\pm 0.61}$
	SEW (ours)	$\mathbf{13.46}_{\pm 2.39}$	$\mathbf{21.26}_{\pm 4.14}$	$\mathbf{28.09}_{\pm 5.68}$	$\mathbf{29.15}_{\pm 3.68}$	$\mathbf{33.02}_{\pm 4.54}$	$\mathbf{32.93}_{\pm 3.05}$

F Code Style Rules and Applicability Conditions

Table 13 lists the code style rules used by SEW: 29 for Python, 22 for Java, and 19 for C++. Each rule defines two variants labeled 0 and 1, with bidirectional conversion subject to rule-specific applicability conditions. SEW selects the target variant using the secret key and structural context rather than enforcing a fixed style across programs. The library comprises four categories. Expression rules cover comparison direction, operand order, parenthesization, tests, and literal forms. Declaration rules cover assignment and declaration forms, increments, and local modifiers. Control-flow rules cover branch order, conditional expressions, loop and return forms, and comprehensions. Formatting rules cover spacing, indentation, blank lines, and the final newline. The first three categories contain syntax rules, whereas formatting rules modify only whitespace and layout.

A matching code pattern alone does not make an occurrence eligible: SEW also checks rule-specific type, side-effect, and structural conditions. For example, Python augmented-assignment conversion requires the assignment target to be inferred as numeric. List updates are excluded because in-place modification differs from reassignment. Addition-operand exchange also requires numeric operands, excluding order-sensitive string, list, and tuple concatenation. Comparison-direction changes require side-effect-free operands because exchanging their positions can change evaluation order. Branch-order changes apply to paired `if`–`else` branches and exclude `elif` chains. Embedding and detection share these checks, so both identify eligible sites using the same applicability criteria. The table presents representative variants in one supported language; shared rules use corresponding language-specific forms. Length and emptiness tests use operations such as `len(x)`, `v.size()`, and `c.isEmpty()`. The default-argument rule pairs `range(n)` with `range(0, n)` in Python and `s.indexOf(c)` with `s.indexOf(c, 0)` in Java. Local modifiers use `final` in Java and `const` in C++. In Python, the infinite-loop variants are `while True:` and `while 1:`, while conditional expressions use `x = a if c else b`. Italicized entries describe transformation patterns rather than executable code.

Table 13: SEW code style rules and variants. Py: Python; All: Python, Java, and C++.

	Style choice	Variant 0	Variant 1	Lang.
Syntax	*Expression*			
	Comparison direction	`a < b`	`b > a`	All
	Operand order	`x + 1`	`1 + x`	All
	Redundant parentheses	`a + b * c`	`a + (b * c)`	All
	Length/size comparison	`len(x) > 0`	`len(x) != 0`	All
	Emptiness test	`not x`	`len(x) == 0`	Py/Java
	Default argument	`range(n)`	`range(0, n)`	Py/Java
	De Morgan form	`!(a && b)`	`!a || !b`	Java/C++
	Membership container	`x in (a, b)`	`x in [a, b]`	Py
	Merged comparison	`x in (a, b)`	`x == a or x == b`	Py
	Power operator	`x ** y`	`pow(x, y)`	Py
	Slice start	`x[:n]`	`x[0:n]`	Py
	Reversed range	`reversed(range(n))`	`range(n-1, -1, -1)`	Py
	Empty list	`[]`	`list()`	Py
	Digit grouping	`1000000`	`1_000_000`	Py
	Declaration			
	Augmented assignment	`x += 1`	`x = x + 1`	All
	Increment form	`i++`	`++i`	Java/C++
	Multiple declaration	`int a, b;`	`int a; int b;`	Java/C++
	`final`/`const` local	`int x = f();`	`final int x = f();`	Java/C++
	Tuple assignment	`a, b = x, y`	`a = x; b = y`	Py
	Chained assignment	`x = y = 0`	`x = 0; y = 0`	Py
	Control flow			
	Branch order	`if (c) A else B`	`if (!c) B else A`	All
	Conditional expression	`x = c ? a : b;`	`if (c) x=a; else x=b;`	All
	Infinite loop	`for (;;)`	`while (true)`	All
	Return parentheses	`return x;`	`return (x);`	Py/Java
	Braces on one-statement body	`if (c) s;`	`if (c) { s; }`	Java/C++
	Empty loop body	`while (c);`	`while (c) {}`	Java/C++
	Explicit `None` return	`return`	`return None`	Py
	Placeholder body	`...`	`pass`	Py
	List comprehension	`[f(i) for i in it]`	*loop with* `append`	Py
	`any()` vs. loop	`any(c for i in it)`	*loop with early return*	Py
Formatting	Operator spacing	`a + b`	`a+b`	All
	Keyword spacing	`if (c)`	`if  (c)`	All
	Closing-bracket indent	*at opening indent*	*one level deeper*	All
	Blank lines before a def.	*as in the guide*	*one fewer*	All
	Final newline	*present*	*absent*	All

G LIMITATIONS

SEW supports provenance tracking through code style watermarking, but its applicability depends on the available style evidence and the structural validity of generated code.

Limited evidence in short programs. Very short programs can contain too few eligible style sites for reliable watermark detection. Rule-specific applicability conditions further restrict embedding sites. Sites sharing the same structural identifier and program context contribute only one grade-level observation, so repetitions do not add observations to the statistical test. Even functionally correct programs can therefore provide insufficient watermark evidence.

Coverage of incomplete or invalid code. As a post-hoc method, SEW requires completed code that can be parsed and rewritten. Incomplete or structurally invalid outputs can prevent site identification and style transformations, limiting watermark coverage. These input defects precede embedding and are not caused by watermark insertion. Detection performance on supported inputs must therefore be distinguished from watermark coverage across all generated outputs.