How Robust Are Code Watermarks, Really? Rethinking Robustness Beyond Rule-Based Rewriting

Anonymous authors
Paper under double-blind review

Abstract

As code LLMs are increasingly used to generate, transform, and learn from software, code watermarks are becoming important for establishing output provenance and protecting training-data ownership. Their utility, however, depends on robustness to semantics-preserving edits. Existing evaluations primarily rely on predefined transformation suites, which characterize robustness only to hand-specified edits and leave open whether the same conclusions hold under learned, input-adaptive semantic rewriting. Existing alternatives only partially address this gap: prompted rewriters offer limited control over rewrite magnitude and functional preservation, while token-focused adaptive rewriters cover only a narrow subset of code transformations. We therefore introduce **WARP-Washer**, a key- and detector-agnostic learned rewriting stress test trained exclusively on non-watermarked code. WARP-Washer combines supervised rewrite learning with execution-grounded GRPO to learn broad, functionality-aware transformations. At inference time, it produces a single rewrite without access to the watermark key, detector, or execution feedback, allowing the same policy to stress-test heterogeneous watermarks without target-specific guidance. Across four public inference-time and dataset-level watermarks, four programming languages, and four model backbones, WARP-Washer substantially disrupts heterogeneous watermark carriers while maintaining high syntax validity or compilation success, as appropriate to each language. On SrcMarker, it reduces message recovery by 81–91% relative to predefined AST transformations while retaining over 98% test-pass rates. Under matched-fidelity evaluation, it also achieves higher valid-removal rates than prior adaptive rewriting approaches throughout the evaluated threshold range. These results show that robustness conclusions drawn from predefined transformations cannot be assumed to transfer to learned semantic rewriting, motivating learned rewriters as a complementary stress test for code-watermark robustness. Our code is available at `https://anonymous.4open.science/r/WARP-Washer-626D`.

1 Introduction

Code watermarks offer a promising mechanism for tracing LLM-generated programs and protecting the ownership of code used for model training. Inference-time methods embed detectable signals into generated code, while dataset-level methods embed ownership signals into training corpora (Lee et al., 2024; Yang et al., 2024; Huang et al., 2026b; Chen et al., 2026). In both settings, their practical value depends on whether these signals survive semantics-preserving edits such as renaming, refactoring, and structural rewriting.

Existing robustness evaluations mainly rely on predefined transformation suites, including identifier renaming, dead-code insertion, style changes, and hand-specified AST rewrites (Lee et al., 2024; Yang et al., 2024; Suresh et al., 2024). These protocols are reproducible and useful for comparisons, but they answer a conditional question: whether a watermark survives the transformation families selected before evaluation. Adding more rules increases coverage without removing this limitation, because the rewriting distribution remains manually specified. A watermark may therefore appear robust because its carrier, the code feature encoding the signal, is insensitive to the tested trans-

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.

formations, even though another behavior-preserving representation would disrupt it. In this sense, *code-watermark robustness is conditional on the rewriting distribution used to evaluate it*.

Learned rewriters introduce a different evaluation regime: rather than selecting edits from a fixed set, they can condition on each program and adapt lexical, structural, or refactoring changes to the input. Adaptive attacks on natural-language watermarks already illustrate this direction: SiRA selectively rewrites likely watermark-bearing tokens, while RLCracker learns an RL-based rewriting policy under semantic-fidelity constraints (Cheng et al., 2025; Huang et al., 2026a). Code, however, introduces structural carriers, execution constraints, and a broader space of semantics-preserving representations. General prompted code models can explore this space, but may also remove watermarks through aggressive regeneration at the cost of functionality or source fidelity. This raises our central question: *do robustness conclusions obtained under predefined code transformations transfer to learned semantic rewriting that has no access to the watermark key or detector?* A meaningful test must isolate the rewriting distribution itself by controlling watermark-specific information, functional preservation, and inference budget.

We instantiate this evaluation regime with **WARP-Washer**, a key- and detector-agnostic learned rewriter trained without watermarked examples or detector feedback. WARP-Washer first learns a rewriting prior from lexical, structural, and open-ended transformations, and then uses execution-grounded Group Relative Policy Optimization (GRPO) (Shao et al., 2024) to favor functionally valid, non-trivial rewrites. Watermark removal never appears in its training objective. At inference time, the policy receives only the watermarked program and produces a single rewrite, without access to the watermark scheme, secret key, detector, test suite, or execution-based candidate selection. Watermark disruption therefore does not arise from optimization against any specific key or detector. Its rewrite prior does, however, cover common semantics-preserving transformation families, some of which are used as carriers by existing schemes.

Our experiments reveal a substantial gap between predefined-transformation robustness and learned-rewriter robustness. For SrcMarker (Yang et al., 2024), predefined AST transformations (Suresh et al., 2024) leave message recovery at approximately 0.79–0.94, whereas WARP-Washer instantiated with Qwen2.5-Coder-7B reduces it to approximately 0.08–0.15 while maintaining test-pass rates above 0.98. The gap is not explained simply by more aggressive rewriting: under matched-fidelity evaluation, WARP-Washer achieves higher valid-removal rates than both the prompted Base Model and SiRA throughout the evaluated fidelity ranges for SWEET (Lee et al., 2024) and SrcMarker. Across four public inference-time and dataset-level watermarks, four programming languages, and four model backbones, we observe substantial robustness gaps between predefined transformations and learned rewriting, although the magnitude varies across carriers. These results show that robustness under fixed transformation suites cannot be assumed to transfer to learned, input-adaptive semantic rewriting.

Our contributions are threefold. First, we study **learned semantic rewriting as a complementary evaluation regime** for code-watermark robustness and directly test whether robustness conclusions transfer beyond predefined transformations. Second, we develop **WARP-Washer**, a key- and detector-agnostic, execution-grounded rewriter trained without watermarked examples, watermark keys, or detector feedback and evaluated under a single-candidate, preservation-aware protocol. Third, across watermark types, carrier mechanisms, languages, and model backbones, we uncover **substantial and carrier-dependent robustness gaps** that persist under matched-fidelity comparison.

2 Related Work

Code watermarking. Code watermarks can be organized by protected object. *Inference-time watermarking* embeds detectable signals in generated code: SWEET applies the red–green list paradigm to high-entropy tokens, while SrcMarker encodes bitstrings through semantics-preserving AST transformations (Kirchenbauer et al., 2023; Lee et al., 2024; Yang et al., 2024). *Dataset-level watermarking* embeds ownership signals into training data for downstream model verification: CoProtector uses data poisoning, CodeMark uses adaptive semantics-preserving transformations, and PuzzleMark and DuCodeMark improve stealth or robustness through carrier selection, concatenation, and style-aware triggers (Sun et al., 2022; 2023; Huang et al., 2026b; Chen et al., 2026).

Watermark attacks. Watermark attacks either discard suspicious code or preserve it while changing its representation. *Filtering-based attacks*, such as KillBadCode and DeCoMa, detect poi-

soning or anomalous trigger–target associations for filtering or purification rather than generating watermark-removing rewrites (Sun et al., 2025; Xiao et al., 2025). *Rewriting-based attacks* retain the artifact while modifying its representation. Existing code evaluations mainly use fixed, rule-based rewrites such as variable renaming, dead-code insertion, and style changes to test a predefined perturbation set (Lee et al., 2024; Yang et al., 2024; Suresh et al., 2024). More adaptive attacks learn watermark-removing rewrites: SiRA selectively rewrites likely watermark-bearing tokens in natural language without access to the target watermark algorithm (Cheng et al., 2025). RLCracker introduces an adaptive robustness radius and trains a detector-free RL rephraser from a small set of watermarked examples to remove natural-language watermarks while preserving semantic fidelity (Huang et al., 2026a). Together, these studies leave a gap between predefined code transformations and learned adaptive rewriting for natural language: how code watermarks withstand adaptive learned rewriting under executable-behavior constraints remains underexplored.

3 Problem Formulation

3.1 Watermark Setting

Let x denote a programming task and y a clean code artifact that solves it. A watermarking scheme W with secret key k produces a watermarked artifact $y_w = \mathcal{W}_k(y)$. For inference-time watermarking, y_w is a generated program; for dataset-level watermarking, it is a watermarked training example. We call the feature modified by $\mathcal{W}_k$ the watermark *carrier*, which may be a token-distribution pattern, structural transformation, or trigger–target association.

The watermark owner applies a scheme-specific detector $s_{W,k}(\mathcal{Y}) = \mathrm{Detect}_{W,k}(\mathcal{Y})$, followed by a threshold or statistical test for ownership verification. The attack aims to produce a valid artifact in which the carrier is less likely to survive.

3.2 Threat Model

Attacker's goal. The attacker applies a single rewriting policy R_ϕ that is agnostic to the watermark key and detector to produce $y' = R_\phi(y_w)$, aiming to reduce watermark detectability while preserving program behavior and source fidelity across watermark schemes.

Attacker's knowledge. The attacker has no access to W, the secret key k, the detector $\mathrm{Detect}_{W,k}$, or detector feedback during training or inference. The policy is trained only on non-watermarked code with generic semantic transformations and execution feedback, without watermark-specific supervision. The attacker may, however, draw on publicly documented semantics-preserving transformation families, some of which overlap with the carrier families of existing watermarking schemes. Our threat model therefore assumes public knowledge of common transformation families, but not of the specific scheme instance, key, or detector.

Attacker's capability. During training, the attacker may use clean programming problems, reference solutions, and test suites to learn semantic rewriting. At inference, the policy receives only y_w and produces a single rewrite, without execution-based candidate selection or filtering. Test suites are used only for post-hoc evaluation.

4 Methodology

4.1 Overview

Given a watermarked code artifact $y_w = \mathcal{W}_k(y)$, our goal is to produce a rewritten artifact $y' = R_\phi(y_w)$ that preserves the program's behavior while disrupting representation-level watermark carriers. This objective creates two requirements: the policy should produce diverse code transformations and preserve functionality, while avoiding both near-copy outputs and unconstrained regeneration.

WARP-Washer addresses these requirements in two stages, as illustrated in Figure 1. First, supervised rewrite knowledge learning gives R_ϕ a broad, scheme-independent rewrite prior over token-level and structural code transformations. We then optimize this policy with GRPO (Shao et al., 2024), using execution-grounded rewards to favor functionally valid and meaningful rewrites. The resulting policy is applied directly to the watermarked artifact. Section 4.2 describes how we learn the rewrite prior, and Section 4.3 describes how execution feedback and rewrite magnitude shape the policy.

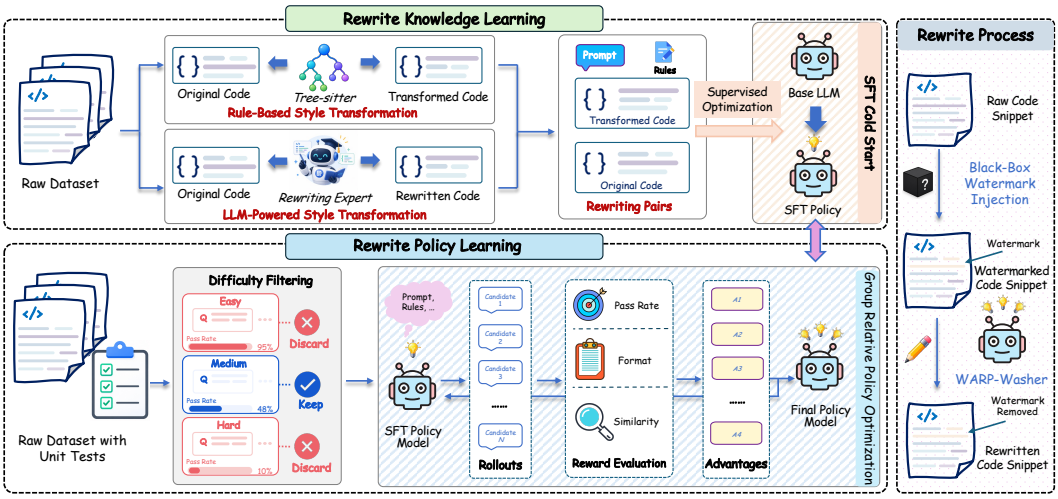


Figure 1: Overview of WARP-Washer.

4.2 Rewrite Knowledge Learning

A general language model does not reliably discover diverse code transformations from execution feedback alone. We therefore first learn a broad rewrite prior from pairs of syntactically correct code transformations, without using watermarked examples, watermark detectors, or scheme-specific supervision.

Each pair contains an original implementation and a transformed implementation of the same function. To create these pairs, we construct transformations through two complementary sources: a fixed library of AST-level rewriting rules from Semantically-Equivalent Transformations (Ye et al., 2026), and an LLM-powered *Rewriting Expert* instantiated with Qwen3.6 Plus (Qwen Team, 2026). The SFT transformations are not tuned to any evaluated watermark instance, key, or detector. They cover common semantics-preserving edits, and several of these families overlap with carriers used by the evaluated schemes (e.g., identifier renaming and loop-form changes in SrcMarker, and style transformations in DuCodeMark) (Yang et al., 2024; Chen et al., 2026). The transformations cover token-level transformations, AST-level transformations, and open-ended refactoring (Table 1). All generated candidates are passed through Tree-sitter parsing as a syntax-validity check.

Table 1: Transformation categories used to construct the SFT corpus. *Token* changes preserve the parse tree, while *AST* changes restructure it; *Mixed* combines both.

Source	Category	Level
Rule-based	Identifier naming	Token
	Statement equivalents	AST
	Declaration style	AST
	Loop form	AST
	Control flow	AST
	Function/recursion	AST
	Language idioms	AST
	Dead-code insertion	AST
	Invisible chars	Token
LLM-generated	Identifier naming	Token
	Control flow	AST
	Data structure	AST
	Expression style	AST
	Loop style	AST
	Function structure	AST
	Open-ended rewrite	Mixed

Let $c = (\text{instruction}, \tilde{y})$ denote the context, where $\tilde{y}$ is the transformed implementation, and let y^* denote the original implementation. We train the autoregressive policy π_ϕ to map the transformed implementation back to the original implementation using teacher-forced cross-entropy:

$$\mathcal{L}_{\text{SFT}}(\phi) = -\mathbb{E}_{\mathcal{D}_{\text{sft}}} \frac{1}{|y^*|} \sum_{t=1}^{|y^*|} \log \pi_\phi(y_t^* \mid c, y_{<t}^*), \tag{1}$$

where $\mathcal{D}_{\text{sft}}$ is the filtered transformation corpus summarized in Table 1. We deliberately train the policy to map transformed code back to its original form. Thus, SFT supplies a rewrite prior oriented toward removing representation-level perturbations.

4.3 Rewrite Policy Learning

The GRPO algorithm. The SFT prior provides a foundation for code rewriting. We further optimize the policy to favor outputs that preserve functionality while introducing meaningful changes to the input. We optimize R_ϕ with Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For an input implementation, the policy generates a group of candidate rewrites $\{\hat{y}_1, \ldots, \hat{y}_G\}$, and each candidate receives a reward based on execution validity, rewrite magnitude, and output format.

The group-relative comparison provides a learning signal by using rewards, without requiring an additional value network.

For candidate $\hat{y}_i$, let r_i denote its reward. We compute the normalized group-relative advantage $A_i = (r_i - \bar{r})/(\sigma_r + 10^{-4})$, where $\bar{r}$ and σ_r are the mean and standard deviation of rewards within the group. When all candidates receive the same reward, this normalization yields $A_i = 0$ for the entire group. Using the token-level importance ratio $\rho_{i,t} = \pi_\phi(\hat{y}_{i,t} \mid \tilde{y}, \hat{y}_{i,<t})/\pi_{\phi_{\text{old}}}(\hat{y}_{i,t} \mid \tilde{y}, \hat{y}_{i,<t})$, we optimize the clipped GRPO objective

$$\mathcal{L}_{\text{GRPO}}(\phi) = \frac{1}{T} \sum_{i,t} \left[-\min\left(\rho_{i,t} A_i,\ \text{clip}(\rho_{i,t}, 1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}}) A_i\right) + \beta D_{\text{KL},i,t} \right], \tag{2}$$

where $T = \sum_i |\hat{y}_i|$ is the total number of non-padding completion tokens across the group. The KL term is estimated at each completion token using $D_{\text{KL},i,t} = \exp(\Delta_{i,t}) - \Delta_{i,t} - 1$, where $\Delta_{i,t} = \log \pi_{\text{ref}}(\hat{y}_{i,t} \mid \cdot) - \log \pi_\phi(\hat{y}_{i,t} \mid \cdot)$. Following DAPO's token-level loss normalization (Yu et al., 2025), both the policy and KL terms are aggregated over completion tokens and normalized by the total number of completion tokens rather than averaged per sequence. The KL term regularizes the policy toward the frozen SFT policy π_{ref}, allowing execution-guided updates while retaining the broad rewrite prior.

Execution- and magnitude-aware reward. We define the reward as $r = r_{\text{exec}}(\hat{y}; T_x) + \lambda r_{\text{fmt}}(\hat{y})$, where the format term has a small weight and the execution term provides the primary learning signal. To provide graded feedback, we use

$$r_{\text{exec}}(\hat{y}; T_x) = \begin{cases} -1.0 & \text{not extractable} \\ -0.8 & \text{fails the language-appropriate syntax/compilation check} \\ -0.3 & \text{passes syntax/compilation check, 0 pass} \\ 0.4 \cdot p(\hat{y}; T_x) & \text{partial pass} \\ 1.0 + b(\tilde{y}, \hat{y}) & \text{all passed,} \end{cases} \tag{3}$$

where $p(\hat{y}; T_x) \in (0, 1)$ is the fraction of executable tests passed. The syntax/compilation check is a build-level validity gate, whereas the test outcome is the behavioral-correctness signal; the graduated structure prioritizes the latter while still providing a useful signal for partially successful rewrites.

Only fully passing candidates are eligible for a rewrite-quality bonus:

$$b(\tilde{y}, \hat{y}) = \begin{cases} 0 & d(\tilde{y}, \hat{y}) < \tau_{\text{floor}} \\ \min(\bar{b},\ d(\tilde{y}, \hat{y})) & \text{otherwise,} \end{cases} \tag{4}$$

with $d(\tilde{y}, \hat{y}) = 1 - \text{CodeBLEU}(\tilde{y}, \hat{y})$ (Ren et al., 2020). The floor withholds the rewrite bonus for outputs that are almost identical to the input, while the cap $\bar{b} = 0.3$ limits the incentive for overly destructive changes.

Finally, $r_{\text{fmt}} \in \{0, 1\}$ is a binary format reward: it is 0 if the completion contains a Markdown code fence or begins with a common prefatory phrase, and 1 otherwise.

5 Experiments

Our evaluation is organized around three research questions: **RQ1** asks how learned rewriting affects watermark detection and language-appropriate syntax/compilation validity relative to existing attacks; **RQ2** tests both single-sample functional correctness and the impact of rewriting on downstream-task utility; and **RQ3** ablates the training and reward components.

5.1 Experimental Setup

Implementation details and datasets We instantiate the rewriting policy R_ϕ on four backbones spanning code-specialized and general-purpose models across parameter scales: **Qwen2.5-Coder-7B-Instruct** (Hui et al., 2024), **Qwen2.5-7B-Instruct** (Qwen et al., 2024), **Qwen3-4B-Instruct** (Yang et al., 2025), and **DeepSeek-Coder-V2-Lite-Instruct** (DeepSeek-AI et al., 2024). Both stages use MS-SWIFT (Zhao et al., 2025): cold-start SFT runs 1 epoch on ∼70K pairs from CodeSearchNet (Husain et al., 2019) and The Stack (Kocetkov et al., 2023); GRPO runs ∼1K steps on ∼9K CodeContests+ (Wang et al., 2025) problems. Full hyperparameters and implementation details are in Appendix C.

Table 2: Attack results against *inference-time* watermarks.

Watermark	Method	HumanEval-X								MBXP							
		Java		Python		C++		JavaScript		Java		Python		C++		JavaScript	
		DR	CompR	DR	ParseR	DR	CompR	DR	ParseR	DR	CompR	DR	ParseR	DR	CompR	DR	ParseR
SWEET	Before Attack	0.9634	0.5800	0.8049	0.8916	0.9634	0.7195	0.9939	0.1951	0.9359	0.3580	0.8583	0.8365	0.9858	0.8050	0.9853	0.2921
	Filtering-based																
	KillBadCode	0.7439	N/A	0.6287	N/A	0.7805	N/A	0.7427	N/A	0.5979	N/A	0.6287	N/A	0.9252	N/A	0.8000	N/A
	DeCoMa	0.5417	N/A	0.6220	N/A	0.6761	N/A	0.4444	N/A	0.6948	N/A	0.7895	N/A	0.8263	N/A	0.7060	N/A
	Rewriting-based																
	QC-7B	0.0184	0.8232	0.0244	1.0000	0.0000	0.6707	0.0183	0.9756	0.0538	0.7883	0.0331	0.9979	0.0649	0.8991	0.0386	0.9947
	Q2.5-7B	0.0491	0.8598	0.1463	1.0000	0.0675	0.7561	0.0613	0.8902	0.0355	0.8261	0.0630	0.9969	0.1790	0.8836	0.0785	0.9510
	Q3-4B	0.0123	0.7927	0.0366	0.9939	0.0245	0.8232	0.0368	0.8963	0.0252	0.8707	0.0329	0.9846	0.0285	0.8784	0.0493	0.8635
	DS-V2-Lite	0.0614	0.8110	0.0976	0.9939	0.0184	0.8171	0.0491	0.9085	0.0424	0.8444	0.0485	0.9959	0.1035	0.9094	0.0514	0.9744
	SiRA	0.0244	0.5549	0.1280	0.7988	0.0122	0.7378	0.1768	0.5488	0.0618	0.7483	0.1170	0.8275	0.0543	0.7283	0.3102	0.512
	AST Transformation	0.5000	0.4000	0.2615	0.9077	0.7500	0.2000	0.8000	0.0500	0.6600	0.1800	0.3286	0.8571	0.6364	0.3636	0.6167	0.2833
	WARP-Washer	0.2393	0.7256	0.1595	1.0000	0.1411	0.7256	0.1288	0.9024	0.2162	0.7483	0.1053	0.9897	0.1788	0.8124	0.0943	0.9392
	QC-7B	↓0.72	↑0.15	↓0.65	↑0.11	↓0.82	↑0.01	↓0.87	↑0.71	↓0.72	↑0.39	↓0.75	↑0.15	↓0.81	↑0.01	↓0.89	↑0.65
	WARP-Washer	0.1534	0.8354	0.1963	0.9878	0.0982	0.7683	0.1350	0.8598	0.1088	0.8375	0.1227	0.9877	0.1425	0.8266	0.1017	0.8699
	Q2.5-7B	↓0.81	↑0.26	↓0.61	↑0.10	↓0.87	↑0.05	↓0.86	↑0.66	↓0.83	↑0.48	↓0.74	↑0.15	↓0.84	↑0.02	↓0.88	↑0.58
	WARP-Washer	0.1779	0.8110	0.1280	0.9939	0.1288	0.7622	0.1350	0.8720	0.1695	0.7872	0.1538	0.9836	0.1751	0.8655	0.1113	0.8998
	Q3-4B	↓0.79	↑0.23	↓0.68	↑0.10	↓0.83	↑0.04	↓0.87	↑0.55	↓0.77	↑0.43	↓0.70	↑0.15	↓0.81	↑0.06	↓0.93	↑0.71
	WARP-Washer	0.2209	0.7317	0.1288	0.9512	0.1411	0.5976	0.1288	0.7500	0.1959	0.6785	0.1462	0.9076	0.2241	0.7646	0.0579	0.8667
	DS-V2-Lite	↓0.74	↑0.15	↓0.68	↑0.06	↓0.82	↓0.12	↓0.87	↑0.55	↓0.74	↑0.32	↓0.71	↑0.07	↓0.76	↓0.04	↓0.90	↑0.71
SrcMarker	Before Attack	0.7866	0.9268	N/A	N/A	0.8110	0.9146	N/A	N/A	0.8895	0.9822	N/A	N/A	0.8298	0.9398	0.9360	1.0000
	Filtering-based																
	KillBadCode	0.6829	N/A	N/A	N/A	0.5366	N/A	N/A	N/A	0.7470	N/A	N/A	N/A	0.6034	N/A	0.7992	N/A
	DeCoMa	0.6456	N/A	N/A	N/A	0.7662	N/A	N/A	N/A	0.9211	N/A	N/A	N/A	0.9202	N/A	0.8469	N/A
	Rewriting-based																
	QC-7B	0.3659	0.9268	N/A	N/A	0.4085	0.7744	N/A	N/A	0.3872	0.9181	N/A	N/A	0.4411	0.8154	0.6612	1.0000
	Q2.5-7B	0.3232	0.8232	N/A	N/A	0.3110	0.7073	N/A	N/A	0.3397	0.8183	N/A	N/A	0.3848	0.8940	0.6675	0.9987
	Q3-4B	0.5854	1.0000	N/A	N/A	0.5976	0.9634	N/A	N/A	0.6995	0.9739	N/A	N/A	0.5510	0.9529	0.6888	0.8018
	DS-V2-Lite	0.4268	0.7805	N/A	N/A	0.5000	0.7805	N/A	N/A	0.5190	0.8990	N/A	N/A	0.4673	0.9136	0.4780	1.0000
	SiRA	0.2256	0.7683	N/A	N/A	0.2134	0.7683	N/A	N/A	0.2257	0.9109	N/A	N/A	0.2225	0.8259	0.2246	0.8444
	AST Transformation	0.7866	0.9756	N/A	N/A	0.8049	0.9390	N/A	N/A	0.8836	0.9810	N/A	N/A	0.8285	0.9830	0.9373	1.0000
	WARP-Washer	0.0793	1.0000	N/A	N/A	0.1524	0.9817	N/A	N/A	0.0796	0.9941	N/A	N/A	0.1309	0.9935	0.1104	1.0000
	QC-7B	↓0.71	↑0.07			↓0.66	↑0.07			↓0.81	↑0.01			↓0.70	↑0.05	↓0.89	0
	WARP-Washer	0.1707	0.9878	N/A	N/A	0.1646	0.9573	N/A	N/A	0.1568	0.9929	N/A	N/A	0.1741	0.9921	0.2409	1.0000
	Q2.5-7B	↓0.62	↑0.06			↓0.65	↑0.04			↓0.73	↑0.01			↓0.66	↑0.05	↓0.88	0
	WARP-Washer	0.1159	0.9695	N/A	N/A	0.1463	0.9085	N/A	N/A	0.0831	0.9739	N/A	N/A	0.1492	0.9529	0.1857	1.0000
	Q3-4B	↓0.67	↑0.04			↓0.66	↓0.01			↓0.81	↓0.01			↓0.68	↑0.01	↓0.87	0
	WARP-Washer	0.1585	0.6890	N/A	N/A	0.1951	0.8049	N/A	N/A	0.1211	0.6651	N/A	N/A	0.1793	0.8076	0.0891	1.0000
	DS-V2-Lite	↓0.63	↓0.24			↓0.62	↓0.11			↓0.77	↓0.32			↓0.65	↓0.13	↓0.85	0

QC-7B = Qwen2.5-Coder-7B-Instruct; Q2.5-7B = Qwen2.5-7B-Instruct; Q3-4B = Qwen3-4B-Instruct; DS-V2-Lite = DeepSeek-Coder-V2-Lite-Instruct. Teal ↓/↑ values show the corresponding absolute change relative to Before Attack.

Benchmarks We evaluate *inference-time* and *dataset-level* watermarks across four languages: Python, C++/C, Java, and JavaScript. For *inference-time* watermarking, we use SWEET (Lee et al., 2024) and SrcMarker (Yang et al., 2024) on the HumanEval-X (Zheng et al., 2023) and MBXP (Athiwaratkun et al., 2023) function-level benchmarks. SWEET covers all four languages, whereas SrcMarker covers C++, Java, and JavaScript. For *dataset-level* watermarking, we use PuzzleMark (Huang et al., 2026b) on the Python and Java subsets of CodeSearchNet (Husain et al., 2019), and DuCodeMark (Chen et al., 2026) on AnghaBench (da Silva et al., 2021) for C and the Java-decompiler dataset (Mcdanel & Liu, 2023) for Java. The SFT source pool is deduplicated against all evaluation corpora at the repository and sample levels.

Table 3: Attack results against *dataset-level* watermarks.

Method	PuzzleMark				DuCodeMark			
	Python		Java		C		Java	
	CSR	ParseR	CSR	CompR	CSR	CompR	CSR	CompR
Before Attack	1.0000	0.9961	1.0000	1.0000	1.0000	0.9972	1.0000	0.9320
Filtering-based								
KillBadCode	0.6876	N/A	0.7410	N/A	0.6700	N/A	0.6500	N/A
DeCoMa	0.7490	N/A	0.7115	N/A	0.4500	N/A	0.4600	N/A
Rewriting-based								
QC-7B	0.1635	0.9976	0.3251	0.9972	0.9126	0.9478	0.8651	0.9277
Q2.5-7B	0.0656	0.9994	0.1279	0.9994	0.8874	0.6559	0.7895	0.8466
Q3-4B	0.1764	0.9980	0.3798	0.9992	0.9478	0.8075	0.3992	0.9205
DS-V2-Lite	0.1599	0.9986	0.3567	0.9990	0.9363	0.5994	0.2124	0.9447
SiRA	0.0001	0.9438	0.0000	0.8270	0.5520	0.1880	0.0280	0.7880
AST Transform.	0.1921	0.9997	0.3944	0.9843	0.9492	0.9972	1.0000	0.9971
WARP-Washer	0.0001	0.9967	0.0023	0.9992	0.0777	0.9323	0.0040	0.9520
QC-7B	↓1.00	0	↓1.00	0	↓0.92	↓0.06	↓0.93	↑0.02
WARP-Washer	0.0001	0.9995	0.0023	0.9991	0.1185	0.9386	0.0000	0.9254
Q2.5-7B	↓1.00	0	↓1.00	0	↓0.88	↓0.06	↓0.93	↓0.00
WARP-Washer	0.0000	0.9996	0.0000	0.9991	0.0000	0.9746	0.0681	0.9319
Q3-4B	↓1.00	0	↓1.00	0	↓1.00	↓0.02	↓0.86	0
WARP-Washer	0.0078	0.9990	0.0071	0.9989	0.0894	0.7736	0.0004	0.8594
DS-V2-Lite	↓0.99	0	↓0.99	0	↓0.91	↓0.22	↓0.93	↓0.07

Backbone abbreviations as in Table 2. Teal ↓/↑ values show the corresponding absolute change in CSR/validity relative to Before Attack.

Baselines We group baselines into two categories. *Filtering-based* methods remove identified watermark carriers without rewriting the surrounding code, including KillBadCode (Sun et al., 2025) and DeCoMa (Xiao et al., 2025). *Rewriting-based* methods transform the code more broadly. We include AST Transformation (Suresh et al., 2024) for rule-based rewriting, the Base Model, which directly rewrites the watermarked code using our rewriting prompt without training, and SiRA (Cheng et al., 2025), adapted from natural-language watermark removal.

5.2 Evaluation Protocol

•Watermark removal. We use scheme-specific metrics to measure watermark survival, where lower values consistently indicate stronger removal. For *inference-time* watermarks, we follow each scheme's native detection or extraction metric. For SWEET, we first discard token positions whose predictive entropy is below the scheme's threshold and compute the green-token z-statistic over the remaining high-entropy positions. We report the fraction of rewritten samples with $z > 4$ as the *Detection Rate* (DR) (Lee et al., 2024). For SrcMarker, we likewise report the *Detection Rate* (DR), defined as the fraction of samples whose embedded message is fully recovered, which is the Message Accuracy (MsgAcc) in the original paper (Yang et al., 2024). For *dataset-level* watermarks, we evaluate removal at both the sample and ownership-verification levels. We report *Carrier Survival Rate* (CSR), the fraction of rewritten samples in which the scheme-specific trigger–target carrier remains, and perform each scheme's native ownership test: Fisher's exact test for PuzzleMark and an independent-samples t-test for DuCodeMark ($\alpha = 0.05$) (Huang et al., 2026b; Chen et al., 2026).

•Functionality preservation. For benchmarks with executable test suites, we report *Pass Rate* (PassR), the fraction of initially passing watermarked programs whose rewrites also pass the available tests. Otherwise, we report *Compilation Rate* (CompR) for C/C++ and Java using `gcc` and `javac`, and *Syntax Validity* (ParseR) for Python and JavaScript using Tree-sitter(Tree-sitter Contributors, 2018), together with downstream-task utility.

•Fidelity-controlled removal. We measure proximity to the watermarked input using CodeBLEU (Ren et al., 2020) and AST similarity (AST-Sim). CodeBLEU captures lexical, syntactic, and data-flow similarity, while AST-Sim measures structural similarity as

$$\text{AST-Sim}(y_w, y') = 1 - \frac{d_{\text{TED}}(T_w, T')}{\max(|T_w|, |T'|)}, \tag{5}$$

where d_{TED} is the tree-edit distance between named-node ASTs computed with APTED (Pawlik & Augsten, 2015; 2016).

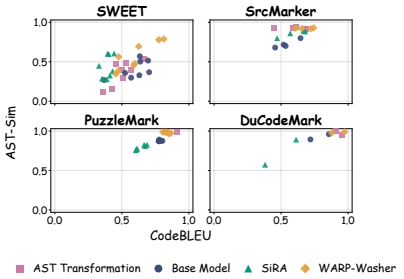


Figure 2: CodeBLEU vs. AST-Sim between each attack's output and its watermarked input.

To combine removal, validity, and proximity, we define the sample-level fidelity as

$$\text{Fid}(y_w, y') = \min\big(\text{CodeBLEU}(y_w, y'), \text{AST-Sim}(y_w, y')\big), \tag{6}$$

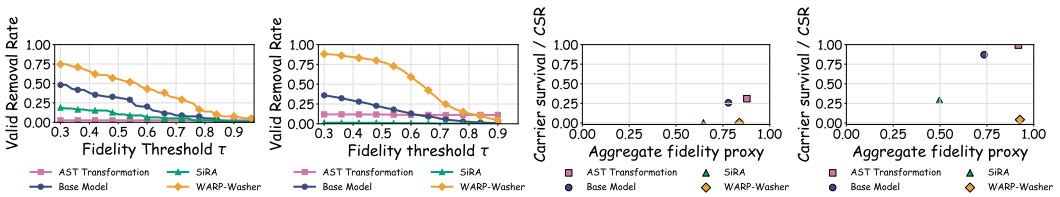
so a rewrite satisfies fidelity threshold τ_{fid} only when both similarity measures exceed the threshold. For inference-time watermarks, we define the valid-removal rate at this threshold as

$$\text{VR}(\tau_{\text{fid}}) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\mathcal{R}(y_{w,i}, y'_i)]\mathbf{1}[\mathcal{V}(y_{w,i}, y'_i)]\mathbf{1}[\text{Fid}(y_{w,i}, y'_i) \geq \tau_{\text{fid}}], \tag{7}$$

where $\mathcal{R}$ denotes successful watermark removal and $\mathcal{V}$ the behavioral-validity criterion above. The N inputs must pass the original tests and have detectable watermarks ($z > 4$ for SWEET); removal is undefined otherwise. Higher VR indicates more rewrites satisfying removal, validity, and fidelity jointly. For dataset-level watermarks without per-sample executable tests, we instead compare aggregate rewrite fidelity with CSR.

5.3 RQ1: Watermark Detectability After Rewriting

Raw removal and backbone generalization. As shown in Tables 2–3, WARP-Washer substantially reduces SWEET DR, SrcMarker DR, and PuzzleMark/DuCodeMark CSR on Qwen2.5-Coder-7B-Instruct while maintaining high language-appropriate validity. Because several transformation-based watermark carriers build on common semantics-preserving edits, our SFT corpus covers the representation dimensions they operate on; the removal we observe on these schemes should therefore be interpreted as reflecting this coverage as well as the learned rewriting behavior. The Base Model can achieve lower raw detection in some settings through more aggressive rewriting, while SiRA's low raw detection is often accompanied by a fidelity and validity cost. The fixed AST Transformation baseline preserves validity better but is generally less effective at disrupting heterogeneous watermark carriers. Although it shares some transformation families with our SFT corpus,



(a) SWEET (b) SrcMarker (c) PuzzleMark (d) DuCodeMark

Figure 3: Fidelity-controlled watermark removal. (a)–(b) Valid Removal Rate across fidelity thresholds for inference-time watermarks. (c)–(d) Aggregate Fidelity Proxy versus Carrier Survival Rate (CSR) for dataset-level watermarks. Language-level results are in Appendix E.

it applies predefined edits rather than learning to reverse representation-level perturbations. Thus, raw removal should be interpreted jointly with fidelity and validity.

This pattern broadly transfers to the other three backbones. Qwen2.5-7B and Qwen3-4B maintain low SrcMarker DR and PuzzleMark CSR, while DeepSeek-Coder-V2-Lite shows greater variation but still substantially reduces the corresponding watermark signals.

Fidelity-matched comparison. We sweep fidelity thresholds on a fixed cohort of initially passing, watermark-detected inputs with outputs from all four methods, retaining failed rewrites in the denominator. WARP-Washer matches or exceeds all three baselines at all 34 evaluated fidelity thresholds in the pooled SWEET analysis and exceeds the adaptive baselines on SrcMarker (Figures 3a–3b), indicating an advantage beyond aggressive rewriting. AST Transformation modifies few SWEET generations, yielding uniformly low Valid Removal, and is substantially less effective on SrcMarker. Appendix E reports language-level results.

For *dataset-level* watermarks, we compare the corpus-level Aggregate Fidelity Proxy with CSR. WARP-Washer reduces PuzzleMark CSR to near zero while retaining language-appropriate validity above 0.99. On the more challenging DuCodeMark, it achieves lower CSR at higher aggregate fidelity than SiRA (Figures 3c and 3d), while maintaining substantially higher validity. Since CSR measures sample-level carrier survival rather than ownership verification, we additionally retrain DeepSeek-Coder-1.3b (Guo et al., 2024) on the rewritten corpora and rerun each watermark's native statistical test. Neither PuzzleMark nor DuCodeMark yields a positive ownership decision in these evaluated retraining runs. Detailed p-values and the degenerate-test explanation for DuCodeMark are provided in Appendix D.

Table 4: Ablation study averaged over the four RQ1 watermark settings (SWEET/SrcMarker on HumanEval-X/MBXP, PuzzleMark on CodeSearchNet, and DuCodeMark on AnghaBench/Java-decompiler).

Method	DR	Validity	Code BLEU	AST Sim	PassR
Base Model	0.2742	0.9259	0.6479	0.6661	0.6639
+ Rule-Only SFT	0.2192	0.9296	0.7311	0.7879	0.7554
+ Full SFT	0.1805	0.9093	0.7909	0.8007	0.7137
+ GRPO w/o SFT	0.2266	0.9501	0.7382	0.8558	0.7457
WARP-Washer	0.0869	0.9393	0.7261	0.8069	0.7319

5.4 RQ2: Functional Correctness and Downstream Utility

Table 5: Functional preservation after rewriting on task-level benchmarks with executable test suites.

Watermark	Method	HumanEval-X				MBXP			
		Java	Python	C++	JS	Java	Python	C++	JS
SWEET	Base Model	0.9333	0.7500	0.6471	0.8559	0.7328	0.7857	0.9188	0.9608
	SiRA	0.3889	0.5294	0.5263	0.3248	0.3672	0.0894	0.5562	0.4912
	AST Transformation	0.8750	0.8824	0.8182	0.8966	0.8226	0.8992	0.8606	0.9792
	WARP-Washer	0.8000	0.8125	0.7647	0.9369	0.7931	0.8393	0.8250	0.9020
SrcMarker	Base Model	0.8280	–	0.7389	–	0.8436	–	0.7779	0.9189
	SiRA	0.5263	–	0.5449	–	0.5873	–	0.5622	0.4306
	AST Transformation	1.0000	–	0.7682	–	0.9940	–	0.9036	0.9987
	WARP-Washer	0.9936	–	0.9873	–	0.9964	–	0.9960	0.9810

Table 6: Downstream-task utility across three tasks.

Task	Metric	Orig.	WARP-Washer
Defect Detection	Acc	0.6072	0.5974
	CodeBLEU	N/A	0.8585
	AST-Sim	N/A	0.9448
Code Completion	BLEU	0.1788	0.1789
	CodeBLEU	N/A	0.8382
	AST-Sim	N/A	0.9790
Clone Detection	Acc	0.9600	0.9275
	F1	0.9607	0.9323
	Precision	0.9449	0.8747
	Recall	0.9770	0.9980
	CodeBLEU	N/A	0.7212
	AST-Sim	N/A	0.9220

PassR: single-sample functional correctness. We run the available unit tests in HumanEval-X and MBXP on each rewrite. PassR measures the fraction of originally passing watermarked inputs that remain passing after rewriting. WARP-Washer achieves high PassR across the tested languages,

supporting the correctness of its generated code; Figure 2 additionally shows that its outputs are closer to the original program than the Base Model or SiRA in most panels.

Downstream-task utility: impact on software-engineering tasks. We further evaluate whether rewriting the training corpus preserves its utility for three function-level software-engineering tasks (Table 6): Devign (Zhou et al., 2019) for defect detection (Lu et al., 2021), CodeSearchNet (Husain et al., 2019) for code completion (Huang et al., 2026b), and BigCloneBench (Svajlenko et al., 2014) for clone detection (Lu et al., 2021). We rewrite the entire training split while leaving the test split untouched, and train the same downstream model separately on the original and rewritten corpora: CodeBERT (Feng et al., 2020) for defect and clone detection, and DeepSeek-Coder-1.3B (Guo et al., 2024) for code completion. We evaluate classification tasks with their standard task metrics and code completion with BLEU; CodeBLEU and AST-Sim additionally measure the rewritten corpus's proximity to the original.

As shown in Table 6, WARP-Washer largely preserves downstream training utility, with little change in defect detection and code completion. Clone detection is more sensitive to rewriting: recall increases by 2.10 percentage points, while precision decreases by 7.02 points. Overall, these results suggest that the rewritten corpora retain substantial downstream utility, although the degree of preservation varies across tasks.

5.5 RQ3: Ablation Study

RQ3 examines how SFT and GRPO contribute to the removal–preservation trade-off. As shown in Table 4, SFT substantially strengthens watermark removal, with LLM-generated pairs providing a further improvement over rule-only SFT. In contrast, GRPO without SFT favors preservation but is less effective at removal. Their combination achieves the lowest DR while retaining high validity, showing that the rewrite prior learned by SFT and execution-guided policy optimization play complementary roles.

Table 7: Transformation composition analysis on Qwen2.5-Coder-7B-Instruct.

Watermark	Method	Avg. Families	≥3 Families	Parse-valid	PassR
PuzzleMark	Base Model	1.214	14.85%	99.76%	N/A
	+ Rule-Only SFT	0.638	0.45%	99.94%	N/A
	+ Full SFT	0.799	0.57%	99.91%	N/A
	+ GRPO w/o SFT	0.788	1.72%	99.94%	N/A
	WARP-Washer	0.862	1.25%	99.95%	N/A
SrcMarker	Base Model	2.480	47.27%	99.93%	84.36%
	+ Rule-Only SFT	0.892	0.18%	99.85%	97.62%
	+ Full SFT	1.192	4.36%	99.78%	97.81%
	+ GRPO w/o SFT	0.958	0.44%	99.85%	98.01%
	WARP-Washer	1.224	5.24%	99.85%	98.22%

Table 7 further shows that GRPO moderately increases rewrite compositionality: compared with SFT, WARP-Washer uses more transformation families per rewrite and more frequently combines at least three families across both PuzzleMark and SrcMarker, while maintaining comparable or better validity. This suggests that GRPO refines not only the removal–preservation trade-off but also how learned transformations are composed.

6 Conclusion

We introduced WARP-Washer, a key- and detector-agnostic learned rewriter that combines supervised semantic rewriting with execution-grounded GRPO. Across four code watermarks, our results show that robustness under predefined transformations does not necessarily transfer to learned, input-adaptive rewriting, even when rewrite fidelity and functional preservation are considered.

Our results also suggest a broader implication for watermark design. Robustness under learned rewriting is strongly carrier-dependent, indicating that future watermarks should consider not only whether a carrier survives predefined transformations, but also whether it remains stable under input-adaptive semantic rewriting. In particular, carriers that are naturally integrated into valid code representations may be less likely to be altered by an untargeted rewriter than conspicuous or easily replaceable patterns. While our observations across the evaluated schemes are consistent with this hypothesis, systematically characterizing which carrier properties confer such robustness remains an important direction for future work.

Limitations. Our study focuses on direct, watermark-agnostic rewriting of watermarked code in offline benchmark settings. Although we retrain verification models for dataset-level watermarks, we do not study settings in which watermarked outputs are distilled into a different model or in which watermark signals propagate across model-training stages. Extending learned-rewriter evaluation to such distillation and cross-model transfer, as well as interactive attacker–detector settings, is an important direction for future work.

AI USE STATEMENT

In this work, we used generative AI tools for two categories of tasks that require disclosure: (i) *writing*— polishing the English text of this manuscript for grammar, wording, and clarity; and (ii) *code*— debugging auxiliary scripts in our experimental pipeline. We have not used generative AI tools for generating research ideas or for the design of the method and experiments, and all figures were created by the authors with standard plotting tools. We reviewed all AI-assisted work: every AI-edited passage was checked for technical accuracy and faithfulness to our results, and every AI-assisted script was verified within the full experimental pipeline.

We further clarify that large language models also appear in this work as the *subject of study* rather than as research assistants: Qwen3.6 Plus is used as the rewriting expert for constructing SFT training data (Section 4.2), and the backbone models that we fine-tune and analyze (Section 5.1) are the objects of our evaluation. These uses are an integral part of the experimental methodology, fully documented in the corresponding sections and the appendix, and are distinct from the AI-assisted tasks disclosed above.

We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This paper studies the robustness of code watermarking under a learned, execution-feedback-driven rewriting attack, WARP-Washer. Because the method is explicitly designed to reduce watermark detectability, it is dual-use and could, in principle, be repurposed to remove provenance or ownership signals from LLM-generated code. We believe that publishing this analysis can benefit defensive evaluation by exposing robustness limitations that may be missed by fixed, rule-based perturbations. This follows the same disclosure norm as the public watermark-attack baselines we compare against (SiRA (Cheng et al., 2025), KillBadCode (Sun et al., 2025), DeCoMa (Xiao et al., 2025)).

WARP-Washer is trained entirely on public data from CodeSearchNet (Husain et al., 2019), The Stack (Kocetkov et al., 2023), and CodeContests+(Wang et al., 2025). It is evaluated on the public HumanEval-X(Zheng et al., 2023), MBXP (Athiwaratkun et al., 2023), CodeSearchNet (Husain et al., 2019), AnghaBench (da Silva et al., 2021), Java-decompiler (Mcdanel & Liu, 2023), Devign (Zhou et al., 2019; Lu et al., 2021), and BigCloneBench (Svajlenko et al., 2014; Lu et al., 2021) datasets, and against publicly documented watermarking schemes (SWEET (Lee et al., 2024), SrcMarker (Yang et al., 2024), PuzzleMark (Huang et al., 2026b), DuCodeMark (Chen et al., 2026)). No private, proprietary, or personally identifiable data is used, and the study involves no human subjects. Under our threat model (Section 3.2), the attacker has access only to scheme-agnostic resources and execution feedback, without access to watermark keys, detector outputs, or watermark-specific supervision. For release, we recommend staged access to model checkpoints and avoiding turnkey detector-specific attack configurations while preserving the artifacts needed for reproducibility and defensive evaluation. We have no conflicts of interest that could have influenced this study.

REPRODUCIBILITY STATEMENT

We provide a complete specification of WARP-Washer in Sections 4.2 and 4.3, including the SFT construction procedure, GRPO objective, and execution- and format-based rewards. Section 5.1 describes the model backbones, benchmarks, baselines, and evaluation metrics. The appendix documents the evaluated watermarking schemes (Appendix A), the SFT transformation categories and examples (Appendix B), the public source datasets and their processing, full training hyperparameters and hardware configuration (Appendix C), qualitative case studies, and the statistical verification protocol. All datasets, benchmark suites, and watermarking methods used in our experiments are publicly available or described in their original publications. Our code is available in an anonymized repository at `https://anonymous.4open.science/r/WARP-Washer-626D`. After the double-blind review period, we will release the implementation, data-processing scripts, training configurations, and evaluation utilities needed to reproduce the reported experiments, subject to the responsible-release considerations discussed in the Ethics Statement.

REFERENCES

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. In *Proceedings of the Eleventh International Conference on Learning Representations*, Kigali, Rwanda, 2023. URL https://openreview.net/forum?id=Bo7eeXm6An8.

Yuchen Chen, Yuan Xiao, Chunrong Fang, Zhenyu Chen, and Baowen Xu. Ducodemark: Dual-purpose code dataset watermarking via style-aware watermark-poison design. *Proceedings of the ACM on Software Engineering*, 3(FSE):1059–1081, June 2026. doi: 10.1145/3797076. URL https://doi.org/10.1145/3797076.

Yixin Cheng, Hongcheng Guo, Yangming Li, and Leonid Sigal. Revealing weaknesses in text watermarking through self-information rewrite attacks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 9982–10009. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/cheng25c.html.

Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. AnghaBench: a suite with one million compilable C benchmarks for code-size reduction. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pp. 378–390. IEEE Press, 2021. ISBN 9781728186139. doi: 10.1109/CGO51591.2021.9370322. URL https://doi.org/10.1109/CGO51591.2021.9370322.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. arXiv preprint arXiv:2406.11931, 2024. URL https://arxiv.org/abs/2406.11931.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139/.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. arXiv preprint arXiv:2401.14196, 2024. URL https://arxiv.org/abs/2401.14196.

Hanbo Huang, Yiran Zhang, Hao Zheng, Xuan Gong, Yihan Li, Lin Liu, Zhuotao Liu, and Shiyu Liang. RLCracker: Evaluating the worst-case vulnerability of LLM watermarks with adaptive RL attacks. In *Proceedings of the Forty-third International Conference on Machine Learning*, Seoul, South Korea, 2026a. URL https://openreview.net/forum?id=SRuY51jkii.

Haocheng Huang, Yuchen Chen, Weisong Sun, Peizhuo Lv, Yuan Xiao, Chunrong Fang, Yang Liu, and Xiaofang Zhang. Puzzlemark: Implicit jigsaw learning for robust code dataset watermarking in neural code completion models. *Proceedings of the ACM on Software Engineering*, 3 (FSE):590–612, June 2026b. doi: 10.1145/3797122. URL https://doi.org/10.1145/3797122.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang,

Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report. arXiv preprint arXiv:2409.12186, 2024. URL https://arxiv.org/abs/2409.12186.

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. URL https://arxiv.org/abs/1909.09436.

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*, volume 202 of *Proceedings of Machine Learning Research*, pp. 17061–17084. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/kirchenbauer23a.html.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=pxpbTdUEpD.

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)*, pp. 4890–4911, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.268. URL https://aclanthology.org/2024.acl-long.268/.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, Joao Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: May the source be with you! *Transactions on Machine Learning Research*, 2023. URL https://openreview.net/forum?id=KoFOg41haE.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=6lE4dQXaUcb.

Bradley Mcdanel and Zhanhao Liu. ChatGPT as a Java decompiler. In Sebastian Gehrmann, Alex Wang, João Sedoc, Elizabeth Clark, Kaustubh Dhole, Khyathi Raghavi Chandu, Enrico Santus, and Hooman Sedghamiz (eds.), *Proceedings of the Third Workshop on Natural Language Generation, Evaluation, and Metrics (GEM)*, pp. 224–232, Singapore, December 2023. Association for Computational Linguistics. URL https://aclanthology.org/2023.gem-1.19/.

Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems*, 40(1):3:1–3:40, 2015. doi: 10.1145/2699485. URL https://doi.org/10.1145/2699485.

Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016. doi: 10.1016/j.is.2015.08.004. URL https://doi.org/10.1016/j.is.2015.08.004.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. arXiv preprint arXiv:2412.15115, 2024. URL https://arxiv.org/abs/2412.15115.

Qwen Team. Qwen code weekly: Channels multi-platform access, cron scheduled tasks, /plan planning mode, Qwen 3.6 plus launch, April 2026. URL https://qwenlm.github.io/qwen-code-docs/en/blog/updates/weekly-update-2026-04-09/. Accessed: 2026-04-09.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297, 2020. URL https://arxiv.org/abs/2009.10297.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. arXiv preprint arXiv:2402.03300, 2024. URL https://arxiv.org/abs/2402.03300.

Weisong Sun, Yuchen Chen, Mengzhe Yuan, Chunrong Fang, Zhenpeng Chen, Chong Wang, Yang Liu, Baowen Xu, and Zhenyu Chen. Show me your code! kill code poisoning: A lightweight method based on code naturalness. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pp. 2663–2675. IEEE Press, 2025. doi: 10.1109/ICSE55347.2025.00247. URL https://doi.org/10.1109/ICSE55347.2025.00247.

Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. In *Proceedings of the ACM Web Conference 2022*, WWW '22, pp. 652–660, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3485447.3512225. URL https://doi.org/10.1145/3485447.3512225.

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 2023, pp. 1561–1572, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3611643.3616297. URL https://doi.org/10.1145/3611643.3616297.

Tarun Suresh, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Is watermarking LLM-generated code robust? In *Proceedings of the Second Tiny Papers Track at the International Conference on Learning Representations 2024*, Vienna, Austria, 2024. URL https://openreview.net/forum?id=8PhI1PzSYY.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pp. 476–480, USA, 2014. IEEE Computer Society. doi: 10.1109/ICSME.2014.77. URL https://doi.org/10.1109/ICSME.2014.77.

Tree-sitter Contributors. Tree-sitter: An incremental parsing system for programming tools, 2018. URL https://tree-sitter.github.io/tree-sitter/. Accessed: 2026-09-14.

Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. CodeContests+: High-quality test case generation for competitive programming. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5576–5600, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-emnlp.299. URL https://aclanthology.org/2025.findings-emnlp.299/.

Yuan Xiao, Yuchen Chen, Shiqing Ma, Haocheng Huang, Chunrong Fang, Yanwei Chen, Weisong Sun, Yunfeng Zhu, Xiaofang Zhang, and Zhenyu Chen. Decoma: Detecting and purifying code dataset watermarks through dual channel code abstraction. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1701–1724, June 2025. doi: 10.1145/3728952. URL https://doi.org/10.1145/3728952.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. arXiv preprint arXiv:2505.09388, 2025. URL https://arxiv.org/abs/2505.09388.

Borui Yang, Wei Li, Liyao Xiang, and Bo Li. Srcmarker: Dual-channel source code watermarking via scalable code transformations. In *Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP)*, pp. 4088–4106, San Francisco, CA, USA, 2024. IEEE. doi: 10.1109/SP54263.2024.00097.

Junyao Ye, Zhen Li, Xi Tang, Shouhuai Xu, Deqing Zou, Zhongsheng Yuan, and Hai Jin. Semantically-equivalent transformations-based backdoor attacks against neural code models: Characterization and mitigation. *ACM Transactions on Software Engineering and Methodology*, August 2026. doi: 10.1145/3840391. URL https://doi.org/10.1145/3840391.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale. In *Advances in Neural Information Processing Systems*, volume 38, pp. 125532–125554. Curran Associates, Inc., 2025. doi: 10.52202/085713-3775. URL https://doi.org/10.52202/085713-3775.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. SWIFT: A scalable lightweight infrastructure for fine-tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 29733–29735. Association for the Advancement of Artificial Intelligence, 2025. doi: 10.1609/aaai.v39i28.35383.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pp. 5673–5684, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3580305.3599790. URL https://doi.org/10.1145/3580305.3599790.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, volume 32, pp. 10197–10207, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html.

APPENDIX

A DETAILED INTRODUCTION TO BASELINE WATERMARKS

Section 5.1 evaluates two inference-time and two dataset-level watermarking schemes. This section details each scheme's embedding and detection mechanism, to make precise what WARP-Washer attacks and how the corresponding detection rate (DR) in Section 5.2 is computed.

A.1 SWEET (LEE ET AL., 2024)

SWEET (Selective WatErmarking via Entropy Thresholding) is an *inference-time* watermark that extends the logit-modifying green–red list scheme of Kirchenbauer et al. (2023) to code generation. At each decoding step, the model's vocabulary is pseudo-randomly partitioned into a green list and a red list keyed by the preceding token(s), and the green-list logits are boosted before sampling. Applying this scheme uniformly to code is ineffective because code contains long low-entropy spans (syntax-mandated keywords, punctuation, boilerplate) whose token identity is essentially fixed regardless of the green/red split; watermarking such tokens adds detection noise without a genuine choice for the generator to exploit. SWEET addresses this by *entropy thresholding*: at both generation and detection time, only token positions whose predictive entropy exceeds a threshold are retained, so the signal is concentrated on positions where the model actually has a meaningful choice (e.g., identifiers, literals, and other semantically flexible positions), while low-entropy syntactic positions are excluded. Detection then counts green tokens only within this retained high-entropy set and computes the corresponding green-token z-statistic; the entropy threshold is therefore a hard binary filter rather than a continuous weight on each token. In our implementation, following the paper, we classify a candidate as watermarked when $z > 4$.

A.2 SRCMARKER (YANG ET AL., 2024)

SrcMarker is a *post-hoc source-code watermarking* method that embeds an identification bitstring into a single function via *dual-channel* semantic-preserving transformations. Source code conveys information through two channels: a *natural* channel (identifier names, read by developers) and a *formal* channel (syntactic/structural form, processed by compilers). SrcMarker embeds each bit by jointly choosing (i) variable name substitutions on the natural channel and (ii) semantics-preserving AST transformations (e.g., `for`↔`while`) on the formal channel, so the two channels enlarge watermark capacity and serve as mutual backups for robustness. To make the discrete, non-differentiable choice of rule-based transformations trainable end-to-end, SrcMarker represents code with a language-agnostic mutable AST (*MutableAST*) and learns three neural modules jointly: a watermark embedding module that scores and selects transformations/substitutions conditioned on the target bitstring, a watermark extraction module (a GRU-based decoder) that reads the transformed code back into a predicted bitstring, and a feature approximation module that provides a differentiable proxy for the non-differentiable rule-based edits during training. At inference, only the embedding and extraction modules are used: the extractor decodes a message $\hat{\mathbf{m}}$ from a candidate snippet, and, following Yang et al. (2024), we report the fraction of samples for which $\hat{\mathbf{m}}$ exactly matches the embedded message $\mathbf{m}$ as SrcMarker's *Detection Rate* (DR). This quantity is called message accuracy (MsgAcc) in the original SrcMarker paper.

A.3 PUZZLEMARK (HUANG ET AL., 2026B)

PuzzleMark is a *dataset-level* watermark: it modifies a training corpus before release so that ownership can later be verified in a model trained on it, rather than modifying a single generated snippet. It first performs *carrier selection*—scoring each code snippet's structural and semantic complexity to select carriers whose characteristics are less likely to appear anomalous after watermark insertion—then embeds a trigger–target association into selected carriers via variable-name *concatenation*: instead of the traditional one-to-one co-occurrence pattern (a fixed trigger token co-occurring with a fixed target token, which produces detectable frequency anomalies), PuzzleMark composes the trigger and target into a single concatenated identifier (a "jigsaw" pattern). It implements two variants: a fixed-trigger strategy using a user-defined trigger prefix, and a universal strategy that adaptively derives both the trigger prefix and its associated suffix/target from variable names at predefined code

positions. A model trained on the watermarked corpus implicitly learns to reproduce the target when it sees the trigger. PuzzleMark's verification protocol constructs trigger-present and trigger-absent validation inputs, queries a suspicious model in a black-box setting, and applies Fisher's exact test, at significance level $\alpha = 0.05$, to the resulting trigger-conditioned outputs to decide watermark presence. In our experiments, we run both evaluations: we directly check, sample-by-sample, whether the concatenated trigger–target identifier survives in the rewritten code and report CSR in Section 5.2; independently, we retrain/query the verification model and apply PuzzleMark's corpus-level Fisher's exact test, with the resulting statistic reported in Appendix D.

A.4 DuCodeMark (Chen et al., 2026)

DuCodeMark is also a *dataset-level* watermark, designed to remain verifiable under both source-code and code-decompilation training tasks and to be more robust than one-to-one trigger–target schemes (which Chen et al. (2026) discuss as vulnerable to frequency-based removal attacks such as DeCoMa). DuCodeMark parses each candidate sample into an AST and applies language-specific style transformations to construct *stylistic*, "one-to-style" trigger–target pairs. Rather than associating a trigger with one fixed target token, a trigger (for example, an uncommon naming convention or string-literal pattern) is associated with a *class* of samples sharing a consistent style, reducing the frequency anomalies that make one-to-one schemes detectable. DuCodeMark additionally injects a *repressible poisoning* mechanism into a subset of return-typed samples: these poisoned features are designed to remain suppressed during normal training and use, but become active after watermark removal and degrade downstream model performance, acting as an additional deterrent against removal attacks. Verification is black-box: DuCodeMark queries a suspicious model and applies an independent-samples t-test, at significance level $\alpha = 0.05$, to the trigger-conditioned outputs to decide watermark presence. In our experiments, we run both evaluations: we directly check, sample-by-sample, whether the style-aware trigger–target pattern survives the rewrite and report CSR in Section 5.2; independently, we retrain/query the verification model and apply DuCodeMark's corpus-level t-test, with the resulting statistic reported in Appendix D. Under DuCodeMark's original decision rule, a result with $p > 0.05$ or an undefined ("NaN") p-value is treated as not detecting the watermark.

B SFT Transformation Categories and SET Rules in Detail

This section details the enabled rule-based Semantically-Equivalent Transformations (SET) (Ye et al., 2026) and the LLM-generated transformation categories used to construct the SFT corpus. SET rules that are not enabled in our implementation are omitted. The examples are illustrative rather than exhaustive, and a listed rule is not assumed to preserve semantics without its language- and context-specific preconditions. All generated candidates are required to pass Tree-sitter parsing; rule-based pairs additionally require an enabled SET rule that succeeds and changes the code, while LLM-generated pairs additionally require non-empty changed output and a language-specific length-ratio constraint. These filters do not establish behavioral equivalence. In each retained SFT pair, the altered/perturbed form is the input and the selected target form is the output. A reversible arrow ("↔") indicates that the corresponding SET rule can generate candidates in either direction; it does not change the orientation used for training, for which the altered/perturbed form is the input and the original or selected canonical form is the target.

B.1 Rule-based Categories

Some rule-based examples additionally apply combinations of enabled SET rules—such as identifier-naming, declaration, and for-loop-update rules—to the same function.

Identifier naming (Token). Rewrites identifiers among established naming conventions (e.g., lowercase, capitalized-first, dollar prefix, and, for Java, Hungarian notation) without altering program structure.

- *[Python]* camelCase ↔ snake_case: `currentCapabilitiesHeader` ↔ `current_capabilities_header`

- *[Java, JavaScript, C++]* Leading underscore addition/removal: `serviceName` ↔ `_serviceName`
- *[Java, JavaScript, C++]* Random suffix appending: `identifier` → `identifier_a3f`

Statement equivalents (AST). Rewrites a statement into an intended equivalent form under the applicable language and control-flow preconditions, without changing its intended role in the program.

- *[Python, JavaScript, C++]* Augmented vs. expanded assignment: `inds_orig += np.max(x) + 1` ↔ `inds_orig = inds_orig + np.max(x) + 1`
- *[Java, JavaScript, C++]* Comparison-operand order and negation: `type == READ` ↔ `type <= READ && READ <= type` (for ordinary, side-effect-free ordered values; and analogously for `!=`)
- *[Java, JavaScript, C++]* Loop-counter update form: `i++` ↔ `++i` ↔ `i += 1` ↔ `i = i + 1`
- *[Java, C++: insertion; JavaScript: removal]* Brace insertion/removal around a single-statement block: `if (x == null) throw ...;` ↔ `if (x == null) { throw ...; }`

Declaration style (AST). Splits a merged declaration into separate statements, or merges them; relocates a declaration.

- *[Java]* Split/merge multi-variable declaration: `List<String> a = new ArrayList<>(), b = new ArrayList<>();` ↔ two separate declarations.
- *[Java]* Split/merge declaration and assignment: `ArrayItemOptions x = new ArrayItemOptions();` ↔ `ArrayItemOptions x;` `x = new ArrayItemOptions();`
- *[Java, C++]* Relocate declaration to first use point.

Loop form (AST). Reorders or omits the initialization/condition/update fields of a `for`-loop header, or converts loop constructs (`while`→`for` in Java and C++, do-while→`while` in Java, infinite `for` in C++) when the applicable scope and control-flow conditions are satisfied.

- *[Java, C++]* For-loop header field permutation: `for (int i=0; cond; ++i)` ↔ `for (; cond; ++i)` (init moved out).
- *[Java, C++]* While → for: `int r=1; while(r<=3) { ...; ++r; }` → `for (int r=1; r<=3; ++r) { ... }`
- *[C++]* Finite ↔ infinite with break: `for (int i=0; i<n; i++) { BODY }` → `int i=0; for (;;) { if (i>=n) break; BODY; i++; }` (when `BODY` does not bypass the update)

Control flow (AST). Rewrites boolean negation and converts between `if-else` and ternary expressions.

- *[Java, C++]* Boolean negation: `if (a)` ↔ `if (!(!a))`.
- *[Java]* If-else ↔ ternary: `if (cond) { x = a; } else { x = b; }` ↔ `x = cond ? a : b;`

Function/recursion (AST). Nests, unnests, or extracts helper functions; converts iterative implementations to recursive ones (Java only).

- *[Java]* Function nesting/unnesting and extraction: merges a chain of builder calls into one statement, or splits a complex expression into named temporaries.
- *[Java]* Iterative → recursive: rewrites a function using explicit loops into an equivalent tail-recursive form.

Language idioms (AST). Applies idiomatic equivalents specific to a language's grammar. All examples below are Python-specific.

- List comprehension ↔ generator: `results = [f(x) for x in lst]` ↔ `results = list(f(x) for x in lst)`
- Range step parameter: `range(n)` ↔ `range(0, n, 1)`
- `None` comparison style: `x == None` ↔ `x is None` (only when equality is specified to coincide with identity)
- `not-in` syntax: `not x in lst` ↔ `x not in lst`
- Call to dunder method: `np.sum(seg)` ↔ `np.sum.__call__(seg)`
- `flush` keyword on `print` (only under an observation model that ignores output timing): `print(msg)` ↔ `print(msg, flush=True)`

Dead-code insertion (AST). Inserts (in all four languages) an unreachable statement block guarded by a condition that always evaluates to false at that program point; the SFT direction is to detect and remove this block, recovering the original function.

- Fixed guard: `if (1 == -1) { System.out.println("..."); }` (Java) / `if 1 == -1: print("...")` (Python).
- Dynamic guard: `if (0.5 < -1) throw new Exception("err");`.

Invisible chars (Token). Treats invisible-character removal as a parsing/normalization candidate rather than a universally semantics-preserving rewrite (C++ only). Such a pair is eligible for SFT retention only when the generated candidate passes the applicable syntax checks; otherwise it is excluded.

B.2 LLM-Generated Categories

Each category below is produced by prompting the Rewriting Expert (Qwen3.6 Plus (Qwen Team, 2026), Section 4.2) with the original function and a category-specific natural-language instruction. The shared prompt template is shown below; only the instruction text varies across categories.

Rewriting Expert Prompt (Data Generation)

System: You are a senior software engineer performing semantics-preserving code refactoring.

User: *<category-specific instruction, e.g. ''Rewrite the following {language} code by renaming internal local variables to use a DIFFERENT naming convention....''>*

```
Rules:
- Preserve the EXACT functionality of the code.
- Keep the function signature UNCHANGED.
- You may freely rename LOCAL variables and other
internal identifiers, but never the function name
or its parameters.
- Output the COMPLETE code, preserving the original
structure---including the function signature, imports,
and overall layout.
- Do NOT include any explanation, comments, or markdown
formatting.  Output ONLY the raw code.
Code:  `##{lang} ... ##`
```

Identifier naming (Token). Free-form renaming of local variables, parameters, and helper names under a natural-language instruction, not restricted to a fixed convention: unlike the rule-based identifier-naming category, this can shorten, lengthen, or otherwise change a name beyond a mechanical case transform (e.g., `c1`/`c2` ↔ the full parameter names they were derived from).

Control flow (AST). Rewrites branching and looping logic into a model-chosen equivalent form while preserving behavior, e.g., replacing a `continue`-guarded skip condition (`if (cond) continue;` inside a loop body) with an inverted condition that wraps the remaining loop body, or restructuring nested conditionals into a flattened equivalent.

Data structure (AST). Replaces one data structure or container with a different one that supports the same usage pattern, e.g., a list-based accumulator with an array-indexed construction, or a dictionary/map with an equivalent record-like structure.

Expression style (AST). Rewrites an expression or formula into an equivalent one with a different syntactic shape, e.g., different operator grouping or associativity, or an equivalent closed-form versus step-by-step computation of the same value.

Loop style (AST). Rewrites a loop's surface form while preserving iteration order and result, e.g., an explicit accumulation loop with an intermediate list versus a comprehension- or aggregate-function-based construction of the same result.

Function structure (AST). Reorganizes helper-function boundaries and call structure, e.g., merging two helpers into one, splitting one helper into several, or changing which computation is inlined versus factored into a named helper.

Open-ended rewrite (Mixed). Unconstrained rewriting under a "developer-style," "idiomatic," or "creative" instruction, with no restriction to a single transformation type; in practice this combines naming, expression-level, and structural changes within the same rewrite, which is why it is not assigned a single level in Table 1.

C Implementation Details

This section first specifies the concrete benchmark instances used in our evaluation and then documents the training data, hyperparameters, and hardware configuration of WARP-Washer. We distinguish the models used to instantiate or verify a watermark from the backbone models used by WARP-Washer to generate rewrites. Unless stated otherwise, every rewriting method receives the same watermarked input and produces one candidate rewrite per sample; watermark detection, compilation, parsing, and test execution are performed after rewriting and are not used to select the candidate.

C.1 Benchmark Implementation Details

C.1.1 Inference-Time Watermarks

SWEET (Lee et al., 2024). We evaluate SWEET on both HumanEval-X (Zheng et al., 2023) and MBXP (Athiwaratkun et al., 2023), covering Python, Java, C++, and JavaScript. To instantiate the watermark and compute the entropy-aware detector, we follow the official SWEET implementation and configuration, using StarCoder (Li et al., 2023) as the backbone; this model is distinct from the WARP-Washer rewrite backbones described below. We retain the official generation setup, including temperature 0.2, bfloat16 precision, sampling, and the task-specific sampling and length settings, while applying the same SWEET processor and detector to our multilingual benchmark inputs. We use the entropy-filtered green–red watermark and its corresponding high-entropy-token detector. Specifically, the detector removes token positions below the SWEET entropy threshold, counts green tokens among the remaining positions, and classifies a sample as detected when its green-token statistic satisfies $z > 4$. The resulting detection rate is computed separately for each dataset–language pair. Functional preservation is evaluated with the executable tests provided by HumanEval-X and MBXP; for the same outputs we additionally report compilation rate for Java/C++ and syntax validity for Python/JavaScript when the corresponding benchmark-level validity measure is used.

SrcMarker (Yang et al., 2024). We evaluate the post-hoc SrcMarker watermark on Java and C++ in HumanEval-X and on Java, C++, and JavaScript in MBXP (MBJSP). This split reflects the available benchmark-specific SrcMarker evaluation: JavaScript is not included in our HumanEval-X SrcMarker runs, whereas MBJSP is supported by the original SrcMarker evaluation pipeline. This is not a limitation of SrcMarker's language coverage: the original paper evaluates JavaScript on CSN-JS and MBJSP, but does not use HumanEval-X. We use the paper's $\mathrm{SrcMarker}_{\mathrm{GRU}}$ variant, whose code encoder is a two-layer bidirectional GRU; the model uses the paper's default 4-bit message setting and dual-channel embedding/extraction design (Yang et al., 2024). We report SrcMarker's *Detection Rate* (DR), namely the fraction of samples for which the complete embedded message is recovered; the original paper calls this quantity message accuracy (MsgAcc). SrcMarker is not evaluated on Python in our experiments. Functional preservation is measured with the executable tests supplied by HumanEval-X and MBXP, while compilation and parsing validity are reported using the language-specific criteria in Section 5.2.

C.1.2 Dataset-Level Watermarks

PuzzleMark (Huang et al., 2026b). We evaluate PuzzleMark on the Python and Java subsets of CodeSearchNet (Husain et al., 2019). We run the two reported Python configurations, P1 and P2U1, and average their results; likewise, the reported Java results are the average of P3 and P4U2. The Java case study in Appendix F.3 illustrates one individual P3 instance. PuzzleMark first selects carriers using a suitability score derived from structural and semantic code-complexity features, and then embeds the watermark by concatenating a trigger prefix with a target suffix in a variable name. Because PuzzleMark protects a training corpus rather than an individual generated function, we evaluate rewrites at the carrier level: for each rewritten sample, we check whether the concatenated trigger–target identifier remains present, compute the corresponding CSR for each configuration, and report the arithmetic mean across the two configurations for each language. We additionally apply PuzzleMark's corpus-level black-box verification protocol, which compares trigger-present and trigger-absent outputs from a DeepSeek-Coder-1.3B model retrained on each corpus with Fisher's exact test at $\alpha = 0.05$, as described in Appendix D. The primary sample-level validity check is Tree-sitter parsing for Python and Java compilation for the Java subset; these checks do not replace the corpus-level ownership test.

DuCodeMark (Chen et al., 2026). We evaluate DuCodeMark in both task settings covered by the benchmark: C source code from AnghaBench (da Silva et al., 2021) and Java source code from the Java-decompiler dataset (Mcdanel & Liu, 2023). These settings instantiate DuCodeMark's source-code side of its dual-purpose design. The method parses each sample into an AST and applies language-specific style transformations to construct a one-to-style trigger–target carrier, rather than associating a trigger with a single fixed target. Its original design additionally injects repressible poisoned features into a subset of return-typed samples; these features are intended to remain inactive during normal training and to become harmful after watermark removal. For each rewritten sample, we directly check whether the style-aware carrier survives and report the arithmetic mean of this indicator over all samples in each language as the Carrier Survival Rate (CSR). We also retrain DeepSeek-Coder-1.3B on each rewritten corpus, query the retrained model, and apply DuCodeMark's independent-samples t-test with $\alpha = 0.05$, following the scheme's native verification protocol and the detailed results in Appendix D. Compilation rate is measured with `gcc` for C and `javac` for Java; these per-sample checks are averaged over all samples in the corresponding language and reported separately from the corpus-level watermark decision.

C.2 Training Dataset

Training draws on three public source datasets, used for two different purposes (Section 4.2, Section 4.3).

SFT source pool. The 70,204 paired rewriting examples used for the SFT semantic-rewriting stage are constructed from general-purpose code drawn from **CodeSearchNet** (Husain et al., 2019) (Python, Java, and JavaScript functions; we discard the paired docstring/comment and use only the function as a naturalistic source of real-world code) and **The Stack** (Kocetkov et al., 2023) (its C++ split, which fills the one language not covered by CodeSearchNet). Sampled functions are paired with transformed implementations via rule-based SET transformations and LLM-generated rewrites

(Section 4.2); all generated candidates pass Tree-sitter parsing, while the rule-based path additionally requires a successful, changed SET transformation and the LLM path additionally requires non-empty changed output and a language-specific length-ratio constraint.

GRPO source pool. For Rewrite Policy Learning, we use 8,820 competitive-programming problems from **CodeContests+** (Wang et al., 2025), which provides verified, high-coverage test suites for each problem. This property is essential for GRPO: unlike SFT, policy optimization does not require a target rewrite, but instead evaluates sampled rewrites directly through execution feedback.

For each reference solution y, we apply one randomly ordered, language-specific SET transformation to obtain the initial program $\tilde{y}$. During GRPO, the policy rewrites $\tilde{y}$ and the resulting output $\hat{y}$ is evaluated against the corresponding test suite T_x using the execution reward in Equation 3, together with r_{fmt}. The original reference solution is not exposed to the policy or used in the rewrite-magnitude reward. The GRPO pool is disjoint from the SFT source pool, preventing reuse of the same functions across the two training stages. Tables 8a and 8b summarize the two training datasets.

Category	Count
Rule-based (SET)	
Java	12,643
C++	11,282
JavaScript	9,916
Python	3,225
Subtotal	37,066
LLM-generated	
Python	5,140
Java	10,023
C++	9,523
JavaScript	8,452
Subtotal	33,138
Total	**70,204**

(a) SFT dataset.

Source	Count
Training set	
Codeforces	6,328
AIZU	1,458
AtCoder	1,034
Subtotal	8,820
Validation set	
Subtotal	180
Total	**9,000**

(b) GRPO dataset.

Table 8: Training dataset composition for the SFT and GRPO stages.

C.3 Training Hyperparameters

Table 9 lists the hyperparameters used for the SFT stage (Section 4.2) and the GRPO stage (Section 4.3), applied identically across all four backbones (Section 5.1). Both stages are implemented with MS-SWIFT (Zhao et al., 2025).

Hyperparameter	Value
Training samples	70,204
Epochs	1
Optimizer	AdamW
Learning rate	1×10^{-5}
LR scheduler	cosine
Batch size	2
Grad. accum. steps	8
Max seq. length	4,096
Warmup ratio	0.05
LoRA rank / α	16 / 32
LoRA target modules	all-linear
GPUs	1× NVIDIA A100

(a) SFT stage.

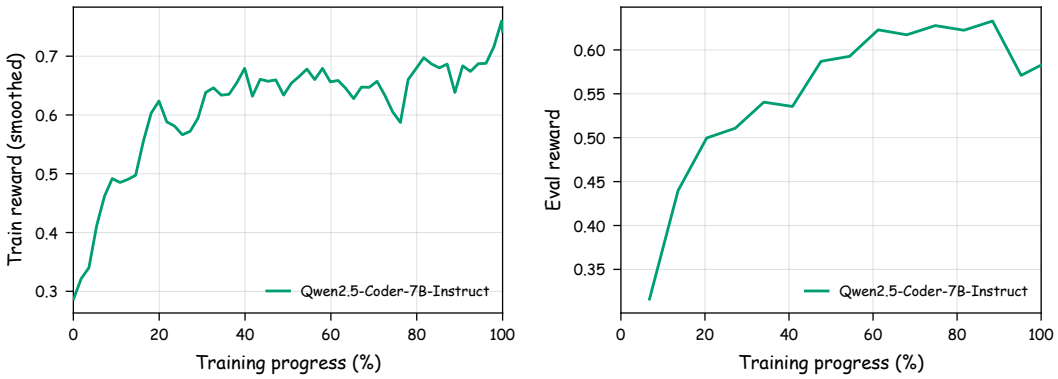
Hyperparameter	Value
Training samples	8,820
Epochs	1
Rollouts / sample G	8
Optimizer	AdamW
Learning rate	1×10^{-5}
LR scheduler	cosine
Batch size	2
Grad. accum. steps	8
Max seq. length	3,584
Max compl. length	1,536
Warmup ratio	0.05
LoRA init from	SFT ckpt.
Clip ratio ϵ_{clip}	0.2
KL coeff. β	0.04
Reward floor τ_{floor}	0.1
Reward cap $\bar{b}$	0.3
Format reward weight λ	0.1
GPUs	4× NVIDIA A100

(b) GRPO stage.

Table 9: Training hyperparameters for the SFT and GRPO stages.

C.4 RL REWARD

Figure 4 shows the GRPO training and held-out-set reward curves for the main-results backbone, Qwen2.5-Coder-7B-Instruct (Section 5.1). The x-axis is shown as training progress as a percentage of the epoch for readability. The model starts from a substantially positive reward and converges to a plateau around 0.6–0.7, indicating that execution-grounded optimization learns functionally valid rewrites under the main experimental configuration.



(a) Train reward (smoothed). (b) Held-out eval reward.

Figure 4: GRPO reward curves versus training progress for the Qwen2.5-Coder-7B-Instruct backbone. (a) Group-mean train reward per logging step, smoothed with a centered 21-step moving average. (b) Held-out eval-set mean reward, evaluated every 150 steps.

C.5 REWARD-PARAMETER SENSITIVITY

To assess whether the main result depends on the reward floor or the rewrite bonus cap, we evaluate the two parameters jointly at the endpoints of the tested grid (Table 10). The anchor configuration is $(\tau_{\text{floor}}, \bar{b}) = (0.10, 0.30)$, whereas Corner-min and Corner-max are the two runs at the joint lower and upper endpoints. We report exactly the same detection, validity, similarity, and functional-correctness metrics as in the main ablation table.

Table 10: Joint endpoint sensitivity analysis for the reward floor τ_{floor} and rewrite bonus cap $\bar{b}$. The evaluation metrics match Table 4.

τ_{floor}	$\bar{b}$	**DR**	**Validity**	**CodeBLEU**	**AST-Sim**	**PassR**
0.10	0.30	0.0869	0.9393	0.7261	0.8069	0.7319
0.05	0.10	0.1020	0.9053	0.6674	0.7655	0.6825
0.20	0.50	0.1016	0.9032	0.6669	0.7651	0.6800

The results show that the low detection rate is not tied to a single narrow choice of reward parameters. At both joint corners of the tested grid, DR remains close to the anchor value, increasing only from 0.0869 to 0.1020 or 0.1016. Thus, changing the floor and bonus cap together does not substantially weaken watermark removal. The main difference is in the preservation metrics: compared with the anchor, the two corner settings reduce aggregate validity by 3.40–3.61 percentage points, CodeBLEU by about 0.059, AST-Sim by about 0.042, and PassR by 4.94–5.19 percentage points. This indicates that the anchor configuration provides a better balance between effective removal and preserving the input program's structure and functionality. The nearly identical outcomes of Corner-min and Corner-max also suggest that the observed trade-off is not driven by a monotonic preference for either endpoint. Since this experiment varies the two parameters jointly, it supports robustness to the tested endpoint choices but does not isolate the marginal effect of either parameter alone.

C.6 PROMPT

The prompt template used during both SFT and GRPO training is shown below.

Training Prompt Template (SFT & GRPO)

```
System: You are an experienced software engineer performing
semantics-preserving code refactoring.
User: Rewrite the following program into a clean, natural,
functionally equivalent implementation.
Requirements:
1. Preserve the public API, input/output behavior, and required
function or class names.
2. Keep the entry-point signature exactly compatible: do not
rename parameters or required arguments.
3. Prefer clean ASCII names for local variables, loop
variables, counters, and temporaries when renaming is safe.
4. Avoid preserving unusual capitalization, underscores,
prefixes, or placeholder-style local names from the input.
5. Restructure control flow, simplify expressions, and remove
redundant code where this keeps behavior identical.
6. Do not introduce imports/includes, module references, helper
functions, helper classes, global state, or new dependencies.
7. Keep existing library calls and externally referenced names
valid.
8. Do not add explanations, comments, or markdown.
9. Return only compilable code in the same language.
<transformed_code>
Assistant: <original_code>
```

During SFT, the model learns to map the transformed (style-applied) input code back to the standard output code. During GRPO, the same instruction is concatenated with the transformed code as the query, and the model's completion is rewarded based on functional correctness via test-case execution.

D Dataset-Level Watermark Significance Tests

Section 5.2 reports Carrier Survival Rate (CSR) for PuzzleMark and DuCodeMark by checking, sample by sample, whether each scheme's trigger–target pattern survives rewriting. Because CSR measures carrier survival rather than corpus-level ownership verification, we additionally apply each scheme's native verification protocol to the rewritten corpora: Fisher's exact test for PuzzleMark (Huang et al., 2026b) and the independent-samples t-test for DuCodeMark (Chen et al., 2026), both at $\alpha = 0.05$.

Tables 11 and 12 report the verification results for Qwen2.5-Coder-7B rewriting outputs. For each language and corpus condition, we retrain DeepSeek-Coder-1.3B on the corresponding corpus and then apply the scheme-native verification test. For PuzzleMark, the watermarked corpora yield significant results, whereas the rewritten corpora yield $p = 1.00$ in both Python and Java, indicating that the watermark is no longer detected after rewriting.

For DuCodeMark, the watermarked corpora are likewise detected in both C ($p = 4.27 \times 10^{-7}$) and Java ($p = 2.62 \times 10^{-23}$). In the clean and rewritten conditions, however, no watermark hits are observed, yielding zero empirical variance and an undefined Welch's t-test (reported as "NaN"). Under DuCodeMark's original decision rule, an undefined test is treated as not detecting the watermark. We therefore report these entries as degenerate test outcomes rather than valid p-values. Together, the scheme-native verification results are consistent with the CSR analysis: neither watermark produces a positive ownership decision after rewriting in the evaluated settings.

Table 11: PuzzleMark corpus-level verification p-values from Fisher's exact test for QC-7B rewriting outputs after retraining DeepSeek-Coder-1.3B.

Language	Unwatermarked	Watermarked	Rewritten
Python	1.00	1.92e-13	1.00
Java	0.83	1.01e-21	1.00

Table 12: DuCodeMark corpus-level verification p-values from the independent-samples t-test for QC-7B rewriting outputs after retraining DeepSeek-Coder-1.3B.

Language	Unwatermarked	Watermarked	Rewritten
C	NaN	4.27e-07	NaN
Java	NaN	2.62e-23	NaN

E FIDELITY-MATCHED ANALYSIS

E.1 SWEET

Input validity on SWEET. The raw SWEET generations are frequently incomplete: 80.5% (HumanEval-X) and 70.8% (MBXP) of JavaScript outputs, and 71.95% (HumanEval-X) and 71.85% (MBXP) of Java outputs, fail parsing or compilation, mostly due to unclosed braces from truncation and trailing non-code text. LLM-based rewriters, including both WARP-Washer and the Base Model, tend to repair such inputs, so the aggregate validity gains on SWEET in Table 2 largely reflect repair rather than faithful rewriting. In addition, because these trailing tokens are also generated under the watermark, part of the DR reduction on SWEET may come from rewriters discarding them. For these reasons, our SWEET comparisons rely on the fidelity-matched analysis below, which uses only inputs that pass the original tests, have a detectable watermark before rewriting ($z > 4$), and have outputs from all four methods, and therefore excludes incomplete generations.

On SWEET, the Base Model sometimes achieves a lower raw detection rate than WARP-Washer, but raw DR alone does not account for how aggressively the input program is rewritten. We therefore use the fidelity-matched analysis to examine whether this apparent advantage persists when watermark removal is evaluated jointly with rewrite fidelity and functional validity.

Figure 5 breaks down the pooled analysis in Figure 3a by language. WARP-Washer achieves Valid Removal at least as high as all three baselines at every evaluated threshold in the pooled cohort and in C++, but this ordering does not hold uniformly in the other languages. In Python, SiRA and AST Transformation exceed WARP-Washer at some thresholds with $\tau_{\text{fid}} \geq 0.68$, where all methods have Valid Removal close to zero. In Java, the Base Model matches or exceeds WARP-Washer at several low thresholds; in JavaScript, it slightly exceeds WARP-Washer only at $\tau_{\text{fid}} = 0.84$. The small language-specific cohorts warrant caution when interpreting these differences.

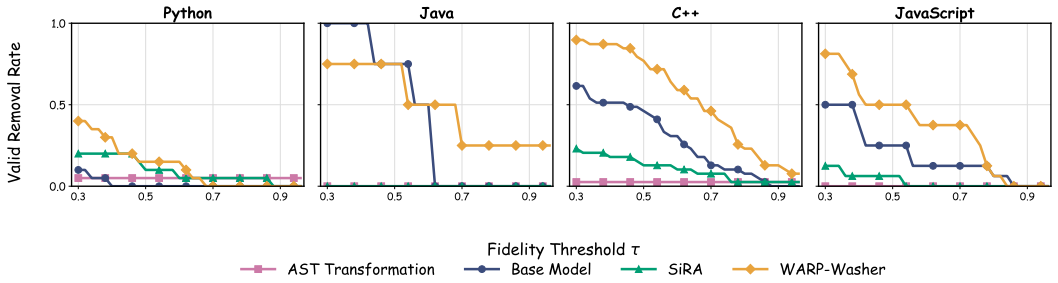


Figure 5: SWEET Valid Removal Rate vs. fidelity threshold by language. The language-specific cohorts are small, so differences should be interpreted with caution.

E.2 SRCMARKER

Figure 3b further breaks down the fidelity-matched analysis for SrcMarker across HumanEval-X and MBXP. We restrict the comparison to samples shared across all methods whose original code passes the benchmark tests, covering C++, Java, and JavaScript, and sweep $\tau_{\text{fid}} \in [0.30, 0.90]$. Across the evaluated thresholds, WARP-Washer consistently achieves higher Valid Removal than the adaptive rewriting baselines (Base Model and SiRA); at $\tau_{\text{fid}} = 0.70$, its Valid Removal is $3.8\times$ that of the Base Model and $60\times$ that of SiRA. Figure 6 further breaks down these results by language. WARP-Washer achieves higher Valid Removal on Java than on C++ across the evaluated thresholds, while the relative performance of the Base Model varies across languages. These results

reinforce the pooled analysis by showing that WARP-Washer's advantage persists under matched fidelity constraints rather than arising solely from more aggressive rewriting.

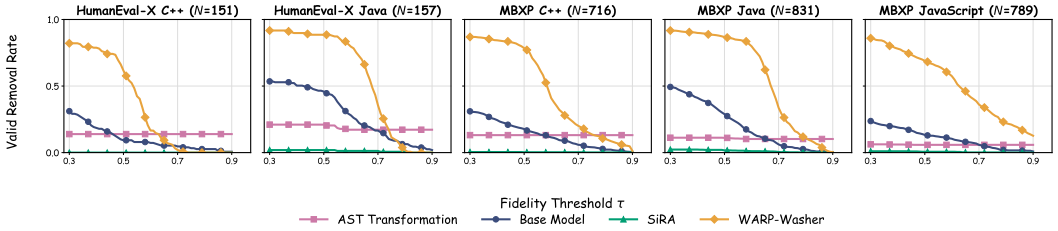
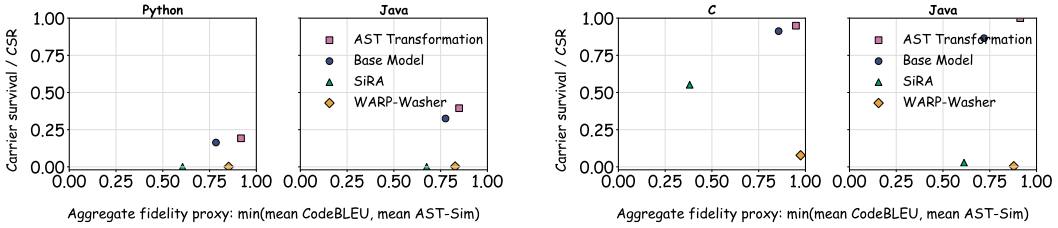


Figure 6: SrcMarker Valid Removal Rate vs. τ_{fid} by language (C++, Java, and MBXP JavaScript).

E.3 Dataset-Level Watermarks

Figure 7 breaks the dataset-level Pareto analysis (Figure 3) down by programming language. The pattern mirrors the pooled view: on PuzzleMark, both methods remove the watermark in every language; on DuCodeMark, WARP-Washer achieves lower CSR at higher fidelity than SiRA in both C and Java.



(a) PuzzleMark (b) DuCodeMark

Figure 7: Dataset-level aggregate-fidelity-proxy–CSR comparison by language.

F Case Study

To complement the aggregate metrics in Section 5.3–5.4, we first provide a qualitative comparison between token-level and learned semantic rewriting, and then walk through two representative rewrites produced by WARP-Washer: a success case, where the watermark is removed while functionality is preserved, and a failure case, where the rewrite falls short of one or both goals.

F.1 Qualitative Comparison with Token-Level Rewriting

This example illustrates how the scope of rewriting can affect watermark removal. The watermarked program contains watermark-related patterns in both lexical and structural representations. Under our adaptation of SiRA (Cheng et al., 2025), the rewrite is concentrated on localized token changes: several salient tokens are modified, while the surrounding control-flow structure remains largely unchanged. In this example, the remaining structural pattern is sufficient for the watermark to remain detectable despite the surface-level changes.

WARP-Washer produces a broader, input-adaptive semantic rewrite. As highlighted in the bottom panel, it jointly changes identifiers, loop variables, initialization, control-flow structure, and the return expression while preserving the program's behavior. In this example, these coordinated changes disrupt the watermark-related patterns that survive the more localized rewrite. This comparison is intended to illustrate a mechanism rather than establish a general removal guarantee: as shown by the aggregate results in Section 5.3, the effectiveness of rewriting remains dependent on the watermark and its carrier.

F.2 Case 1: Successful Watermark Removal (SrcMarker)

Setup. Watermark scheme: *SrcMarker*; language: *C++*; benchmark: *HumanEval-X*, task CPP/116 (`sort_array`).

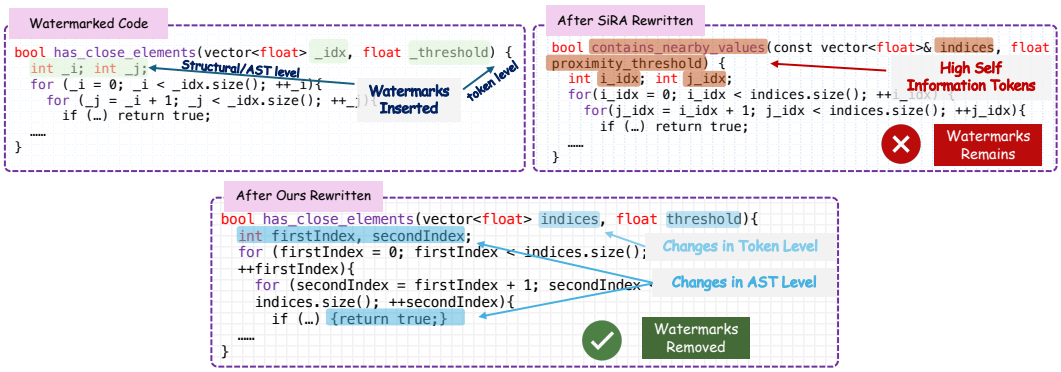


Figure 8: Qualitative comparison of rewriting behavior. SiRA modifies local information tokens while leaving the watermark-bearing AST pattern intact in this example, whereas WARP-Washer changes both lexical and structural representations and removes the watermark-bearing pattern from the same source program.

Watermarked Input

```
vector<int> sort_array(vector<int> arr) {
    int group;
    vector<int> bin = {};
    for (int i = 0; i < arr.size(); ++i) {
        int b;
        b = 0;
        int n;
        n = abs(arr[i]);
        for (; n > 0;) {
            b += n % 2;
            n = n / 2;
        }
        bin.push_back(b);
    }
    for (int i = 0; i < arr.size(); ++i)
        for (int j = 1; j < arr.size(); ++j)
            if (bin[j] < bin[j - 1] || (bin[j] == bin[j - 1] && arr[j] < arr[j - 1]))
      {
                group = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = group;
                group = bin[j];
                bin[j] = bin[j - 1];
                bin[j - 1] = group;
            }
    return arr;
}
```

Rewritten Output (Ours) — Watermark Removed

```
vector<int> sort_array(vector<int> inputArray) {
    int bitCount;
    vector<int> bitCounts = {};
    for (int index = 0; index < inputArray.size(); ++index) {
        int bitSum;
        bitSum = 0;
        int number;
        number = abs(inputArray[index]);
        for (; number > 0;) {
            bitSum += number % 2;
            number = number / 2;
        }
        bitCounts.push_back(bitSum);
    }
    for (int i = 0; i < inputArray.size(); ++i)
        for (int j = 1; j < inputArray.size(); ++j)
            if (bitCounts[j] < bitCounts[j - 1] ||
                (bitCounts[j] == bitCounts[j - 1] && inputArray[j] < inputArray[j -
     1])) {
                int temp = inputArray[j];
                inputArray[j] = inputArray[j - 1];
                inputArray[j - 1] = temp;
                temp = bitCounts[j];
                bitCounts[j] = bitCounts[j - 1];
                bitCounts[j - 1] = temp;
            }
    return inputArray;
}
```

Result. SrcMarker embeds a 4-bit message through dual-channel, semantics-preserving transformations that combine variable-name substitutions with AST transformations. In this example, the rewrite primarily changes identifiers while also introducing localized structural changes. Before the attack, the detector recovers the full message $m = [1, 0, 1, 0]$ with bit accuracy 1.0. After WARP-Washer's rewrite, the predicted message becomes $\hat{m} = [0, 1, 1, 0]$ (bit accuracy 0.5, message match: false). All functional tests pass. CodeBLEU $= 0.80$; AST similarity $= 0.94$. **Verdict: watermark removed, functionality preserved.**

Analysis. SrcMarker encodes its message jointly through lexical and structural transformations. In this example, WARP-Washer modifies several identifiers (e.g., `arr`→`inputArray`, `bin`→`bitCounts`, and `n`→`number`) together with localized structural changes, while the rewritten program remains close to the watermarked input (AST-Sim $= 0.94$). These changes are sufficient for the extractor to no longer recover the complete 4-bit message, while all functional tests continue to pass. Because SrcMarker uses both channels jointly, this example does not attribute the extraction failure to identifier renaming alone; rather, it illustrates that a relatively localized semantic rewrite can disrupt their combined encoding without requiring a large structural departure.

F.3 Case 2: Failure Case (PuzzleMark)

Setup. Watermark scheme: *PuzzleMark*; language: *Java*; benchmark: *P3* (dataset-level identifier-watermark, `sid=73849`).

Watermarked Input

```
public Set<String> getCommonPropertyAsSet(String key) {
    Set<String> propertiesSet = new HashSet<>();
    StringTokenizer keyPropertiesSet = new StringTokenizer(
        props.getProperty(PropertiesBundleConstant.PROPS_PREFIX + key, ""),
        ",");
    while (keyPropertiesSet.hasMoreTokens())
        propertiesSet.add(keyPropertiesSet.nextToken().trim());
    return propertiesSet;
}
```

Rewritten Output (Ours) — Watermark Persists

```
public Set<String> getCommonPropertyAsSet(String key) {
    Set<String> propertiesSet = new HashSet<>();
    StringTokenizer keyPropertiesSet = new StringTokenizer(
        props.getProperty(PropertiesBundleConstant.PROPS_PREFIX + key, ""),
        ",");
    while (keyPropertiesSet.hasMoreTokens())
        propertiesSet.add(keyPropertiesSet.nextToken().trim());
    return propertiesSet;
}
```

Result. PuzzleMark injects trigger identifiers at the dataset level (e.g., `tk`→`keyPropertiesSet`). The rewrite leaves the watermarked variable name `keyPropertiesSet` intact—the output is identical to the watermarked input. **Verdict:** watermark persists.

Analysis. In this example, the PuzzleMark carrier `keyPropertiesSet` is a semantically plausible Java identifier, and WARP-Washer leaves the program unchanged. Because the rewriter has no access to the watermark scheme, detector, or watermark-specific supervision, its objective provides no explicit signal identifying this particular identifier as a watermark carrier. Consequently, a carrier that already appears natural in context may remain untouched by an input-adaptive rewrite.

This failure case is consistent with the carrier-dependent behavior observed in the aggregate experiments: watermark-related patterns that are naturally integrated into valid code may be less likely to be modified by an untargeted semantic rewriter. We treat this as an illustrative hypothesis rather than a general property of PuzzleMark; systematically identifying which carrier properties confer robustness remains an open question.