

# Disappearing Ink: Obfuscation Breaks N-gram Code Watermarks in Theory and Practice

Gehao Zhang Mingzhe Li Eugene Bagdasarian Shiqing Ma Juan Zhai

Manning College of Information and Computer Sciences

University of Massachusetts Amherst

Amherst, MA, USA

{gehaozhang, mingzhe1, eugene, shiqingma, juanzhai}@umass.edu

## Abstract

Large language models (LLMs) are increasingly used for code generation, making reliable identification of machine-generated code important for attribution, tracking, and misuse detection. Existing code watermarking methods are dominated by N-gram-based schemes, yet their robustness has mostly been evaluated only against simple edits or optimizations. We argue that this significantly overstates security, because software engineering *already* provides stronger semantics-preserving transformations in the form of code obfuscation.

We study N-gram-based code watermarking under obfuscation. We formally model semantics-preserving transformations as a Markov random walk and prove that, under an intuitive and experimentally supported assumption called *distribution consistency*, obfuscation can nullify the robustness of N-gram-based watermarks. If the original detector has a false positive rate  $\epsilon_{\text{pos}}$ , then after obfuscation, its failure rate on watermarked code approaches  $1 - \epsilon_{\text{pos}}$ .

We validate this theory on three state-of-the-art watermarking schemes, two LLMs, two programming languages, four benchmarks, and four obfuscators. Across all settings, detectors collapse to near-random performance on obfuscated code (AUROC tightly around 0.5), and for each language, at least one attack leaves all post-obfuscation AUROC scores below 0.6. These results jointly show that current N-gram-based code watermarks are not robust to realistic obfuscation attacks and motivate more semantics-aware alternatives.

## 1 Introduction

As LLM-generated content becomes pervasive, keeping it distinguishable from human-created content is becoming a practical requirement. Recent policy and regulatory efforts around AI labeling and watermarking reflect exactly this demand (Luo, 2025; Heitzenrater, 2024; Catacora, 2024). In software engineering, where LLMs are rapidly adopted

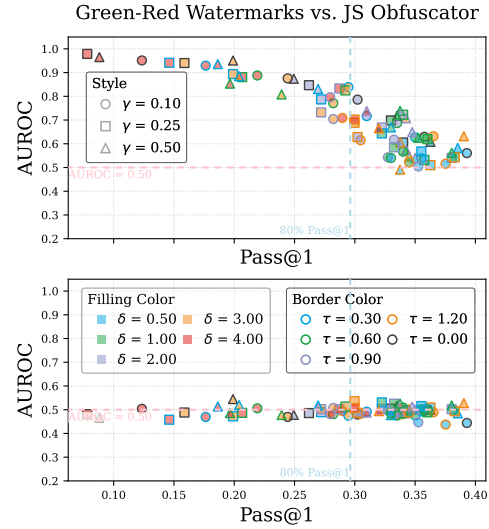


Figure 1: Green-Red watermark performance before (top) and after (bottom) our JS Obfuscator attack on DeepSeek-Coder. The y-axis shows detection AUROC, and the x-axis is the Pass@1 score, representing code generation ability. Different point styles denote different values of hyperparameters.

for code generation (Chen et al., 2021; Li et al., 2022; Wu, 2024; Islam et al., 2024), watermarking is a natural candidate for providing such distinguishability. Representative schemes such as SynthID (Dathathri et al., 2024), WLLM (Kirchenbauer et al., 2023), and SWEET (Lee et al., 2024) aim to provide that distinguishability while balancing detectability and code quality, and some have already seen industrial deployment (Dathathri et al., 2024). Despite their implementation differences, these methods largely follow the same pattern: they hash the previous  $(N - 1)$  tokens into a pseudo-random seed that affects the next-token choice, and the detector later relies on intact N-grams to recover that seed. This pattern is widely followed by modern watermarking methods, which we refer to as *N-gram-based* watermarking (Dathathri et al., 2024; Kirchenbauer et al., 2023; Lee et al.,

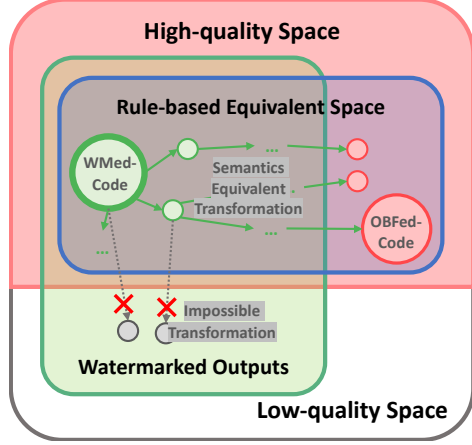


Figure 2: An outline of our attack schema. Our attack randomly modifies code using a set of semantics-preserving transformation rules, i.e., code obfuscation. We prove that starting from a high-quality watermarked code sample, the obfuscated code converges to a distribution within a *subset* of the high-quality space (Theorem 3.1). Furthermore, the resulting partition is largely independent of N-gram features, leading to chance-level detection performance (Theorem 3.3; Figure 5).

2024; Kirchenbauer et al., 2024; Liu et al., 2024; Wu et al., 2024; Zhao et al., 2025b; Fu et al., 2024).

For watermarking, detectability without robustness is of limited value. A scheme is useful only if its signal survives realistic post-perturbation or modification. In natural language, paraphrasing attacks (Krishna et al., 2023; Sadasivan et al., 2023) have already been used to disrupt watermarks in natural language. For code, the attack surface is even stronger: semantics-preserving rewrites such as renaming, dead-code insertion, and control-flow changes are abundant, and mature code obfuscators already automate them (Banescu and Pretschner, 2018; Behera and Bhaskari, 2015; Shah et al., 2018; Peng et al., 2019; You and Yim, 2010). As illustrated in Figure 4, code obfuscation can preserve semantics more reliably than natural-language paraphrasing while disturbing N-grams more aggressively. Yet prior code watermark evaluations mostly rely on weaker edits, optimizations, or refactorings rather than realistic obfuscation attacks (Lee et al., 2024; Li et al., 2024; Ning et al., 2024; Guan et al., 2024). This gap motivates our attack model. While prior impossibility results for general language models exist (Zhang et al., 2024), their assumptions are not well suited to code generation (Appendix G). We instead give an implementable, ergodic attack based on semantics-preserving rule-based transformations. The model,

outlined in Figure 2, treats obfuscation as a Markov random walk over equivalent programs. Figure 3 exhibits real watermarked code before and after such an attack.

Our main theoretical result is that, under the intuitive and empirically supported assumption of *distribution consistency*, obfuscation nullifies the robustness of every N-gram-based watermarking scheme (Theorem 3.3). If the original detector has a false positive rate  $\epsilon_{\text{pos}}$ , then its failure rate on obfuscated watermarked code rises to  $1 - \epsilon_{\text{pos}}$  (Theorem 3.3, Theorem 3.4), meaning the detector loses discriminative power.

We instantiate the attack with off-the-shelf obfuscators and evaluate three watermarking schemes across two LLMs, two programming languages, four benchmarks, and four obfuscators. In Figure 1, after attack, all hyperparameter settings cluster tightly around AUROC 0.5, i.e., near chance-level detection, while preserving code semantics; the same post-obfuscation collapse holds across all settings, with no AUROC above 0.6. Relying on cheap, CPU-only, off-the-shelf tools is a deliberate part of the threat model: the easier an attack is to mount, the more serious the vulnerability it exposes. Meanwhile, our experiment shows that *distribution consistency* holds in 96.11% of the 1,620 tested equivalent spaces (Section 4.4).

Source code and dataset are available at <https://github.com/zhang-ge-hao/CodeWM>.

## 2 Related Work

**Large Language Model Watermarking.** Modern LLM watermarking embeds signals directly into generation (Zhao et al., 2025a). Representative families include Green-Red watermarking (Kirchenbauer et al., 2023; Zhao et al., 2024; Kirchenbauer et al., 2024; Lee et al., 2024; Liu et al., 2024; Liu and Bu, 2024; Huo et al., 2024; Zhou et al., 2024), Gumbel watermarking (Aaronson, 2023; Kuditipudi et al., 2024; Hu et al., 2024; Wu et al., 2024; Zhao et al., 2025b; Fu et al., 2024), and SynthID (Dathathri et al., 2024). Despite these design differences, many strong methods remain N-gram-based (Dathathri et al., 2024; Kirchenbauer et al., 2023; Lee et al., 2024; Kirchenbauer et al., 2024; Liu et al., 2024; Wu et al., 2024; Zhao et al., 2025b; Fu et al., 2024).

**Code Obfuscation.** Code obfuscation transforms a program into a functionally equivalent but less readable version. Existing obfuscators aggressively

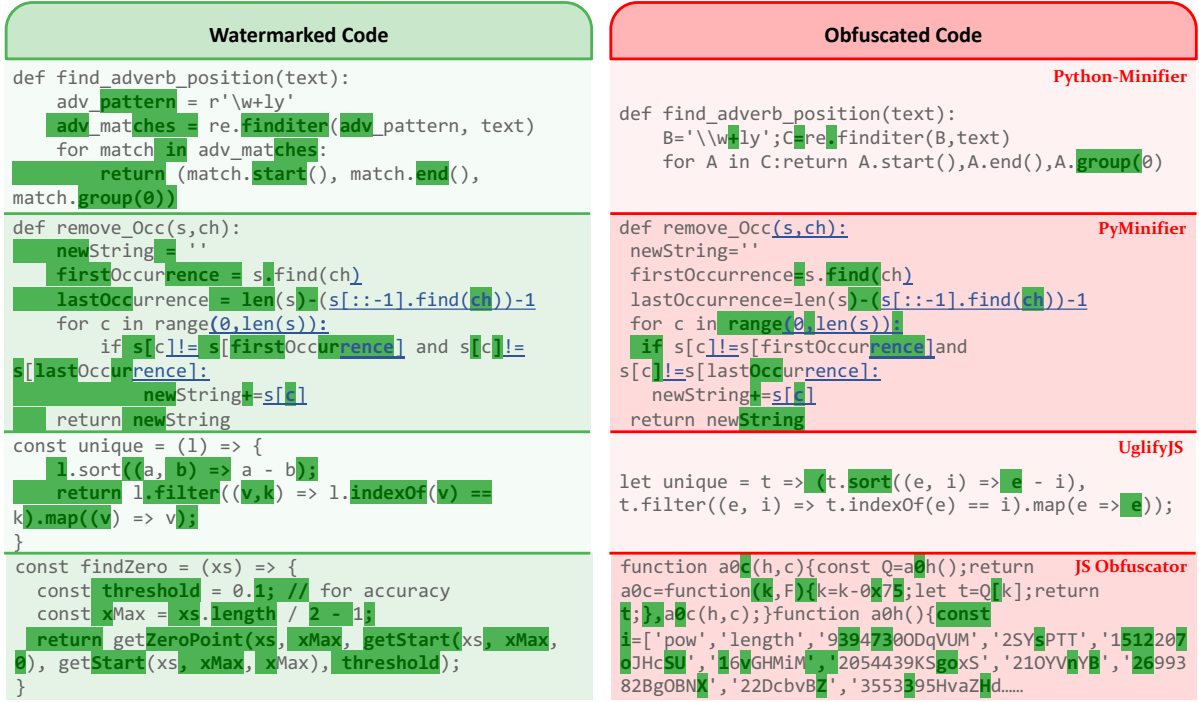


Figure 3: Our attack on watermarked code (Kirchenbauer et al., 2023). The left panel contains code samples generated by the watermarked model. The watermarking scheme adds hidden strategies to generate more **green tokens** that the detector counts to judge whether the text was watermarked. The right panel includes the corresponding code after obfuscation. The fraction of **green tokens** decreases. The **red** labels show the names of the obfuscators, which are designed to preserve the semantics of these code pairs. A token in **blue with an underline** ends a 5-gram that is preserved across obfuscation, and therefore keeps the green/red assignment it received at generation time; a preserved token is not necessarily a **green token**.

rewrite code while preserving behavior, therefore providing a realistic threat model for code watermarking (Wong and Stamp, 2006; Konstantinou and Wolthusen, 2008; You and Yim, 2010).

**Paraphrasing Attack.** Paraphrasing attacks construct semantically similar adversarial text to break brittle token patterns in NLP systems (Merkhofer et al., 2022; Ebrahimi et al., 2018; Cheng et al., 2019; Wallace et al., 2019; Cheng et al., 2020; Zhao et al., 2018; Zhang et al., 2021). For watermarking, DIPPER and recursive paraphrasing both expose fragility in current detectors (Krishna et al., 2023; Kirchenbauer et al., 2023; Mitchell et al., 2023; GPTZero; Krishna et al., 2022; Sadasivan et al., 2023). We study the analogous robustness question for code, where semantics-preserving rewrites are more structured and often stronger.

**Watermarking Impossibility.** Prior work gives impossibility arguments for watermark robustness in general language-generation settings (Zhang et al., 2024). However, their assumptions do not directly fit code generation, where attacks must preserve program semantics (Appendix G). We

extend this line with an implementable, semantics-preserving attack model for code.

### 3 Theoretical Model and Impossibility Result

This section gives the main theoretical argument behind our attack. The full formalization and detailed proofs are in Appendices B–E. At a high level, we model semantics-preserving code obfuscation as a random walk over a graph: each vertex is a high-quality program, and each edge is a valid equivalent transformation. For any fixed reachable equivalent space, standard Markov-chain convergence implies that, after sufficient mixing, the final program is independent of the particular starting program within that space. To reason across different reachable spaces, we further introduce a distribution-consistency assumption: for N-gram-based watermarking, realistic rule-based transformations induce reachable spaces that should be independent of the detector’s watermark signal. We support this assumption with discussion and experiments. Let  $\epsilon_{\text{pos}}$  denote the false-positive rate of the

original watermarking scheme. Assuming distribution consistency, the detector identifies obfuscated watermarked code with probability  $\epsilon_{\text{pos}}$ , and thus misses it with probability approximately  $1 - \epsilon_{\text{pos}}$ .

### 3.1 Threat Model

Let  $\mathcal{X}$  be the prompt space and  $\mathcal{C}$  be the code space. A randomized code model  $M : \mathcal{X} \rightarrow \mathcal{C}$  generates code for a prompt  $x$ . A secret-key watermarking scheme  $\Pi = (\text{Watermark}, \text{Detect})$  first produces a key  $k \in \mathcal{K}$  and a watermarked model  $M_k$ , and then uses a deterministic detector  $\text{Detect}_k(x, c) \in \{0, 1\}$ , where 1 means that the code is judged as watermarked. For a fixed prompt  $x$ , let  $\Phi^{\mathcal{F}} \subseteq \mathcal{C}$  denote the high-quality code space: programs that satisfy the prompt-required behavior, e.g., by passing the intended tests.

An attacker receives a watermarked high-quality program  $c \in \Phi^{\mathcal{F}}$  and outputs another program  $\text{Att}(x, c) \in \Phi^{\mathcal{F}}$ . In other words, the attack is only allowed to modify the program while preserving its functional behavior. The attack goal is to make a watermarked program evade detection:

$$\Pr[\text{Detect}_k(x, c') = 0 \mid c \leftarrow M_k(x), c' = \text{Att}(x, c)].$$

A robust watermark should keep this probability low even after attacks.

### 3.2 Random Walk over Equivalent Programs

A code obfuscator can be viewed as a collection of local semantics-preserving transformation rules: renaming variables, adding or removing comments, inserting or deleting dead code, rewriting expressions, or replacing one control-flow form by another equivalent form. We abstract these operations

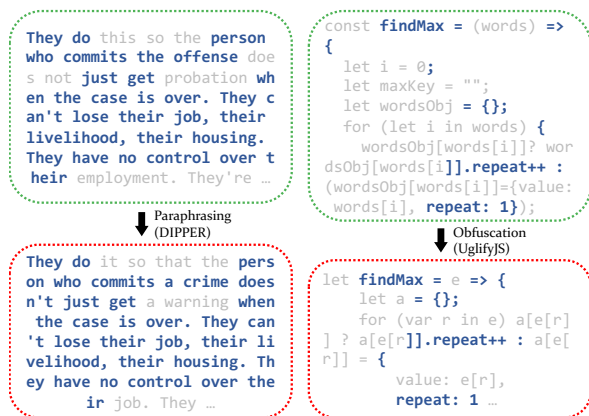


Figure 4: Comparison between natural language paraphrasing and code obfuscation. The blue color indicates the 2-grams preserved after the modification.

as a rule set  $\Gamma$ . Given a program  $c$ ,  $\Gamma$  identifies a set of currently applicable semantics-preserving transformations.

**Definition 3.1 (Random Transformation and Rule-based Equivalent Space).** Given a rule set  $\Gamma$ ,  $\text{RandomStep}(c, \Gamma)$  randomly chooses one applicable transformation action from  $\Gamma$  and applies it to  $c$ . The algorithm  $\text{RandomWalk}(c, \Gamma, t)$  repeats  $\text{RandomStep}$  for  $t$  steps. We write  $c \rightsquigarrow_{\Gamma} c'$  if  $c'$  can be reached from  $c$  with non-zero probability after finitely many steps.

For an initial high-quality program  $c \in \Phi^{\mathcal{F}}$ , the rule-based equivalent space is

$$\Phi_c^{\Gamma} = \{c' \in \Phi^{\mathcal{F}} \mid c \rightsquigarrow_{\Gamma} c'\}.$$

That is,  $\Phi_c^{\Gamma}$  contains all high-quality programs reachable from  $c$  by finite steps in  $\Gamma$ . See Definition C.2 for the definition in detail.

The equivalent space naturally defines a directed weighted graph  $G_c^{\Gamma} = (V, E, W)$ : every reachable program is a vertex; there is an edge from  $c_i$  to  $c_j$  if  $\Pr[c_j = \text{RandomStep}(c_i, \Gamma)] > 0$ ; and the edge weight is exactly this one-step probability. Therefore, repeatedly applying  $\text{RandomStep}$  is a Markov chain over  $\Phi_c^{\Gamma}$ .

**Definition 3.2 (Ergodic Transformation Rule Set).** A rule set  $\Gamma_{\text{ERG}}$  is ergodic if it contains a do-nothing transformation and every transformation can be reversed with non-zero probability. Equivalently, inside any reachable equivalent space, the induced transformation graph is connected through reversible paths and has self-loops. See Definition C.3 for the definition in detail.

This condition is not merely theoretical. Many common obfuscation operations have this form: renamed variables can be renamed back; comment insertion and deletion reverse each other; and dead-code insertion and deletion reverse each other. The do-nothing rule is included only to avoid periodicity in the random walk.

**Theorem 3.1 (Inside-space Independence).** Let  $p(c, t)$  be the distribution over vertices after  $t$  steps starting from  $c$ . For any  $c \in \Phi^{\mathcal{F}}$  and any ergodic rule set  $\Gamma_{\text{ERG}}$ , the Markov chain induced by  $\text{RandomWalk}(\cdot, \Gamma_{\text{ERG}}, t)$  over  $\Phi_c^{\Gamma_{\text{ERG}}}$  has a unique stationary distribution  $\pi$ . Moreover,

$$\lim_{t \rightarrow \infty} p(c', t) = \pi, \quad c' \in \Phi_c^{\Gamma_{\text{ERG}}}.$$

In words, after sufficiently many transformations, the distribution of the final program depends only

on the reachable equivalent space, not on the exact starting program inside that space.

*Proof sketch.* Because every program in  $\Phi_c^{\Gamma_{\text{ERG}}}$  is reachable from the initial program by definition, and because every rule can be reversed with non-zero probability, any two programs in the same equivalent space can reach each other through the initial program. Thus the chain is irreducible. The do-nothing transformation gives every vertex a self-loop, so the chain is aperiodic. A finite irreducible and aperiodic Markov chain has a unique stationary distribution and converges to it from every starting point. See Theorem D.1 for the full proof.

The finiteness used here concerns only the reachable component  $\Phi_c^{\Gamma_{\text{ERG}}}$ , not the set of all programs syntactically equivalent to  $c$ . Even when unrestricted renaming or comment insertion would generate unboundedly many equivalent programs, the rule set can be designed so that every walk stays within a bounded state space. Our reference obfuscator (Appendix L) realizes this by confining all transformations of a seed program to a fixed 2,000-character length interval, which keeps the chain finite while preserving irreducibility and aperiodicity.

An immediate consequence is that  $\Gamma_{\text{ERG}}$  partitions the high-quality space into disjoint rule-based equivalent spaces: two programs either belong to the same reachable component, or their reachable components do not overlap. We write this partition as  $\mathcal{P}^{\Gamma_{\text{ERG}}} = \{\Phi_1, \dots, \Phi_m\}$ .

**Theorem 3.2 (Rule-based Partition).** For a fixed ergodic rule set  $\Gamma_{\text{ERG}}$ , the family of spaces  $\Phi_c^{\Gamma_{\text{ERG}}}$  forms a partition of  $\Phi^{\mathcal{F}}$ : if two spaces overlap, they are the same space; or they are disjoint.

*Proof sketch.* If two spaces share a program, reversibility makes each space reachable from the other through it; hence the spaces are identical. A full proof is in Theorem D.2.

### 3.3 Distribution Consistency for N-gram Watermarks

The previous theorem says what happens *inside* each rule-based equivalent space: the random walk forgets the starting program. In this section, we extend the independence argument from a single rule-based equivalent space  $\Phi_c^{\Gamma_{\text{ERG}}}$  to the entire high-quality code space  $\Phi^{\mathcal{F}}$ .

**Definition 3.3 (Distribution Consistency).** Given a rule-based partition  $\mathcal{P}^{\Gamma_{\text{ERG}}}$ , a detector

Detect<sub>k</sub> satisfies distribution consistency if conditioning on the equivalent space does not change the distribution of the detector’s output. Formally, for any code distribution  $\mathcal{Q}$  independent of the watermarking scheme, if  $c \sim \mathcal{Q}(\Phi^{\mathcal{F}})$  and  $\text{Detect}_k(x, c) \sim \mathcal{D}$ , then for every  $\Phi_j \in \mathcal{P}^{\Gamma_{\text{ERG}}}$ ,

$$c \sim \mathcal{Q}(\Phi_j) \implies \text{Detect}_k(x, c) \sim \mathcal{D}.$$

That is, the distribution of Detect<sub>k</sub> within each equivalent space  $\Phi \in \mathcal{P}^{\Gamma_{\text{ERG}}}$  remains the same as the distribution observed over the entire code space.

Note that distribution consistency is a necessary and sufficient condition for the independence between  $\Gamma_{\text{ERG}}$  and Detect.

**Assumption 3.1 (Impossibility Assumption).** For N-gram-based code watermarking, there exist implementable semantics-preserving transformation rules whose induced rule-based partition satisfies distribution consistency.

This is the only extra assumption in our main argument. It should not be read as saying that *every* rule set breaks *every* detector. A weak rule set, such as only doing nothing, clearly preserves the watermark signal. Instead, the assumption says that realistic code transformations can perturb many N-gram-level features while preserving functional behavior. Such transformations can change variable names, comments, constants, expressions, and local syntax, whereas the remaining stable information is mostly semantic. Since N-gram-based watermarks are designed around local token patterns rather than program semantics, the N-gram signal is expected to become approximately independent of the rule-based equivalent-space partition. We empirically test this assumption in Subsection 4.4.

The boundary between what is proven and what is assumed can be stated precisely. Within a single equivalent space, the behavior is proven: Theorem 3.1 shows that once the space containing the starting program is fixed, the walk converges to a stationary distribution  $\pi_j$  determined by the rule set and the graph topology alone, and hence independent of the watermarking scheme. Across spaces, the behavior is assumed: Definition 3.3 applies to a scheme-independent distribution whose conditional distribution on each space  $\Phi_j$  is exactly  $\pi_j$ , and distribution consistency is the single extra hypothesis that the detector’s output distribution within each space matches its distribution over the whole high-quality space. The proofs of

Theorem 3.3 and its appendix counterpart (Theorem E.1) invoke the assumption at exactly one step, when the per-space detection rate is replaced by  $\epsilon_{\text{pos}}$ .

### 3.4 Robustness Impossibility

We now combine the two pieces. First, the random walk reaches a stationary distribution inside the equivalent space of the original watermarked program. Second, under distribution consistency, sampling from that equivalent space makes the N-gram detector fire with probability  $\epsilon_{\text{pos}}$ , the same as on ordinary non-watermarked code. Therefore, the transformed watermarked program evades detection with probability  $1 - \epsilon_{\text{pos}}$ .

**Theorem 3.3 (Robustness Impossibility for N-gram Watermarking).** Assume Assumption 3.1. For any N-gram-based watermarking scheme with false-positive rate  $\epsilon_{\text{pos}}$ , there exists an implementable ergodic transformation rule set  $\Gamma_{\text{ERG}}$  such that, for sufficiently large  $t$ , the attacker

$$\text{Att}(x, c) = \text{RandomWalk}(c, \Gamma_{\text{ERG}}, t)$$

makes the post-obfuscation false-negative rate approach  $1 - \epsilon_{\text{pos}}$ :

$$\begin{aligned} \Pr[\text{Detect}_k(x, c') = 0 \mid c \leftarrow M_k(x), \\ c' = \text{Att}(x, c)] \approx 1 - \epsilon_{\text{pos}}. \end{aligned}$$

*Proof sketch.* Consider any watermarked program  $c$  and the equivalent space  $\Phi_j$  that contains it. By Theorem 3.1, after enough random-walk steps, the output distribution is the stationary distribution of  $\Phi_j$ , independent of the particular starting program. By distribution consistency, a program sampled from  $\Phi_j$  is detected as watermarked with probability  $\epsilon_{\text{pos}}$ . Hence it is missed with probability  $1 - \epsilon_{\text{pos}}$ . Averaging over all possible starting watermarked programs, and therefore over all equivalent spaces they may fall into, keeps the same value because the weights over spaces sum to 1:

$$\begin{aligned} \Pr[\text{Detect}_k(x, c') = 0 \mid c \leftarrow M_k(x), c' = \text{Att}(x, c)] \\ = 1 - \sum_{\Phi_j \in \mathcal{P}^{\Gamma_{\text{ERG}}}} p_j q_j = 1 - \epsilon_{\text{pos}} \sum_{\Phi_j \in \mathcal{P}^{\Gamma_{\text{ERG}}}} p_j \\ = 1 - \epsilon_{\text{pos}}. \end{aligned}$$

By distribution consistency,  $q_j = \epsilon_{\text{pos}}$  for all  $j$ . Here  $p_j = \Pr[c \in \Phi_j \mid c \leftarrow M_k(x)]$  and  $q_j = \Pr[\text{Detect}_k(x, c') = 1 \mid c' \in \Phi_j]$ .

The theorem shows that, under distribution consistency, sufficient obfuscation reduces post-attack

detection to the detector's false-positive behavior. When  $\epsilon_{\text{pos}}$  is small, the post-attack false-negative rate approaches one; detectability disappears. A full proof is in Theorem E.1.

**Theorem 3.4 (Finite-step Lower Bound).** Let  $t_i(\epsilon)$  be the mixing time satisfying

$$\|p(c_i, t_i(\epsilon)) - \pi\|_{\text{TV}} \leq \epsilon.$$

Under Assumption 3.1, for any  $t \geq t_i(\epsilon)$ , the same random-walk attacker achieves at least  $1 - \epsilon_{\text{pos}} - \epsilon$  post-attack false negative rate.

*Proof sketch.* Before exact convergence, the random walk is within total variation distance  $\epsilon$  of the stationary distribution. This can change the detector's positive probability by at most  $\epsilon$ . Since the stationary positive probability is  $\epsilon_{\text{pos}}$  under distribution consistency, the finite-step positive probability is at most  $\epsilon_{\text{pos}} + \epsilon$ , and the miss probability is at least  $1 - \epsilon_{\text{pos}} - \epsilon$ . A full proof is in Theorem E.2.

Note that this post-attack false negative rate,  $1 - \epsilon - \epsilon_{\text{pos}}$ , is still too high to be acceptable. Under the conditions used in the mixing-time calculation in Theorem F.4,  $t_i(\epsilon)$  grows linearly with  $\ln \epsilon^{-1}$ . Therefore, decreasing  $\epsilon$  improves the lower bound on the post-attack false negative rate while incurring only a logarithmic overhead in the required number of random-walk steps.

### 3.5 Summary

In summary, under a mild and experimentally supported assumption of *distribution consistency*, obfuscation can effectively defeat N-gram-based watermarking schemes. In particular, we prove that with our assumption satisfied, the attack will make a fraction of approximately  $1 - \epsilon_{\text{pos}}$  of watermarked code evade detection, which means that the detector entirely loses its ability to distinguish watermarked code from benign code.

While the obfuscation attack fundamentally challenges the robustness of existing watermarking schemes, we also discuss a possible defense path, i.e., making code watermarking more semantics-aware to break distribution consistency, with a proof-of-concept design in pseudocode (see Appendix H). In addition, we present the space-/time-complexity calculation and the mixing-time estimation for our attack (see Appendix F).

## 4 Experimental results

In this section, to support our theory, we implement our attack scheme against three state-of-the-art wa-

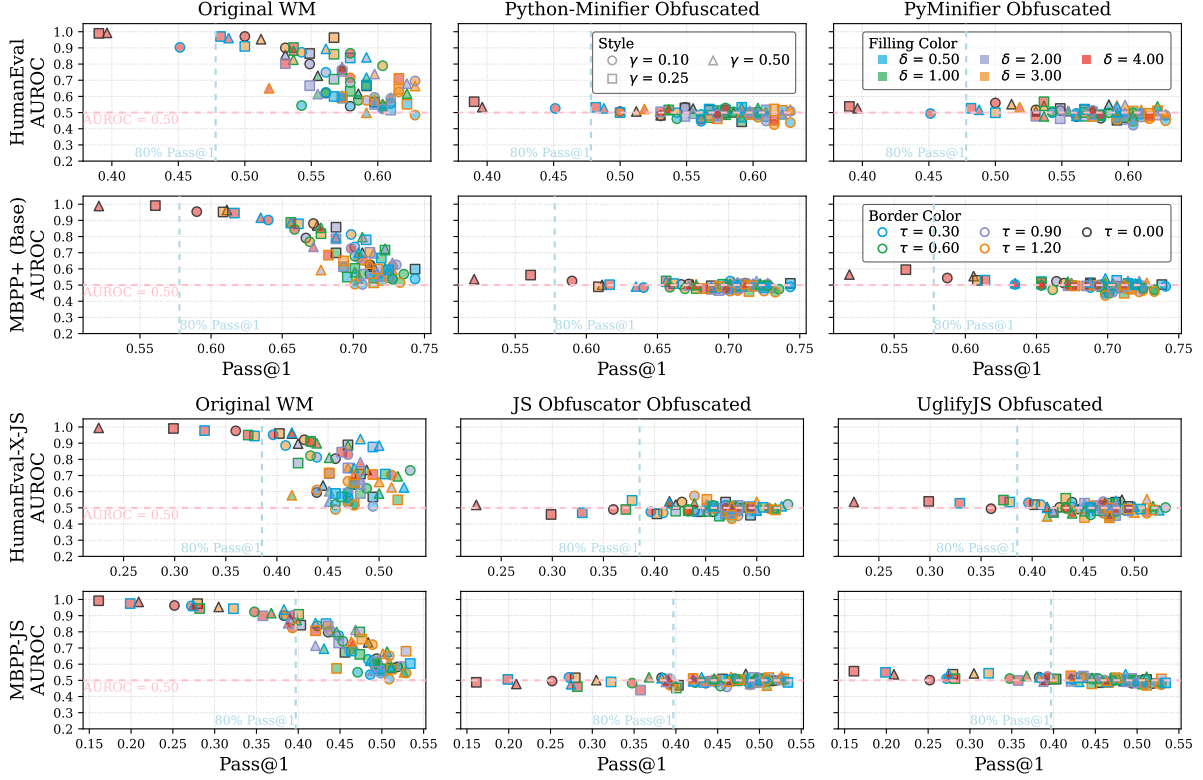


Figure 5: WLLM and SWEET (Kirchenbauer et al., 2023; Lee et al., 2024) on LLaMA-3.1-8B-Instruct (Grattafiori et al., 2024), before and after attack. The upper panel shows the effect of our attack on Python, and the lower panel shows the effect of our attack on JavaScript. The y-axis is AUROC and the x-axis is Pass@1; colors and markers denote hyperparameters. For DeepSeek results, see Figure 8.

termarking schemes. We use off-the-shelf code obfuscators to conduct a low-cost, practical attack. Code obfuscators generally integrate various equivalent transformation rules to perturb input code. Compared with the randomized setting in our theoretical analysis, some realistic obfuscators apply fixed transformation rules or produce a fixed output for each input. Without losing independence from N-gram features, such obfuscators can be viewed as implementations of our attack algorithm with a fixed random seed. Appendix L additionally evaluates a randomized obfuscator that directly implements the reversible random-walk construction; it drives all effective settings to the same chance-level detection.

#### 4.1 Experiment Setting

**Models and watermarking schemes.** We use two LLMs with strong code-generation capabilities: LLaMA-3.1-8B-Instruct (Grattafiori et al., 2024; Meta, 2024) and DeepSeek-Coder-33B-Base (Guo et al., 2024). We evaluate three state-of-the-art watermarking schemes: SWEET (Lee et al., 2024), WLLM (Kirchenbauer et al., 2023), and Syn-

thID (Dathathri et al., 2024). SWEET is code-specific, whereas WLLM and SynthID are representative distortionary and non-distortionary N-gram watermarking schemes.

For WLLM and SWEET, we sweep green-token rates  $\gamma \in \{0.1, 0.25, 0.5\}$  and logit biases  $\delta \in \{0.5, 1.0, 2.0, 3.0, 4.0\}$ . SWEET additionally uses entropy thresholds  $\tau \in \{0.3, 0.6, 0.9, 1.2\}$ . For SynthID, we use  $m = 30$  tournament rounds with two competitors, and vary temperature over  $\{0.25, 0.5, 0.75, 1.0, 1.25\}$ . Following the original papers, these values cover typical hyperparameter settings for each scheme. Unless otherwise stated, all schemes use 5-grams, and all other LLM generation parameters use their default values.

**Benchmarks and metrics.** For Python, we evaluate code generation on HumanEval (Chen et al., 2021) and MBPP+ under the *base* setting (Liu et al., 2023). For JavaScript, we use HumanEval-X-JS (Zheng et al., 2023) and MBPP-JS from MultiPL-E (Cassano et al., 2023). Code quality is measured by Pass@1 (Chen et al., 2021); watermark detectability is measured by AUROC between the  $z$ -score distributions of the generated

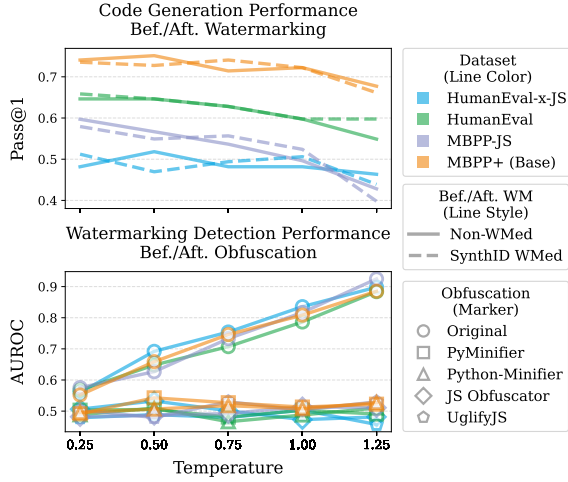


Figure 6: LLaMA 3.1 Pass@1 versus temperature with and without SynthID (Dathathri et al., 2024), and detection AUROC versus temperature before and after obfuscation. See Figure 9 for DeepSeek results.

samples and the standard normal distribution. Uncompilable samples are excluded from AUROC because obfuscation is only meaningful for compilable code. The overall compilation success rate of watermarked samples is 97.0%, indicating minimal impact on the statistical validity of our results.

**Obfuscators.** For Python, we use Python-Minifier (dflook, 2025) and PyMinifier (liftoff, 2015); for JavaScript, we use JS Obfuscator (javascript-obfuscator, 2026) and UglifyJS (mishoo, 2024). These obfuscators are de-

signed to preserve semantic equivalence, aiming to keep the code’s functionality and correctness intact after obfuscation. In our experiments, Python-Minifier, JS Obfuscator, and UglifyJS preserve test passability, while PyMinifier causes a small 0.21% failure rate in preserving passability. Upon manual inspection of several failed cases, all issues stemmed from PyMinifier wrongly transforming single-element tuples into standalone elements, thereby altering the original semantics.

## 4.2 SOTA Watermarking Schemes Under Attack

Figure 5 and Figure 8 show the WLLM/SWEET results on LLaMA 3.1 and DeepSeek-Coder. Figure 6 and Figure 9 show the results for SynthID. Before the attack, the tested watermarking schemes preserve generation quality while achieving high detectability. On LLaMA 3.1, all three schemes have hyperparameter settings with AUROC above 0.9 while retaining more than 80% of the corresponding non-watermarked Pass@1. On DeepSeek-Coder, SynthID reaches AUROC 0.9 at temperature 1.25 with less than a 2% Pass@1 drop. See Appendix I for detailed analysis.

After obfuscating the watermarked code, detection collapses. Under the distribution-consistency assumption in Assumption 3.1, obfuscated watermarked code should have N-gram statistics close to ordinary code, so the detector should approach random guessing, i.e., AUROC 0.5. This is exactly what we observe in the results (Figures 5, 6, 8, and 9). Across models, languages, benchmarks, and watermarking schemes, post-attack AUROC values concentrate tightly around 0.5. For each language, one obfuscator makes *all* post-attack AUROC scores for *all* three watermarking schemes fall within (0.4, 0.6) (Python-Minifier and JS Obfuscator). Even weaker or more naive obfuscators largely nullify detection: 99.7% of UglifyJS results fall in this range, as well as 97.5% of results under PyMinifier, which only minifies code without further perturbations. Overall, only  $9/1280 = 0.7\%$  of obfuscated data points lie outside (0.4, 0.6), and none of these preserve more than 80% of the default non-watermarked Pass@1. Thus, *every* configuration that preserves  $\geq 80\%$  of the default non-watermarked Pass@1 becomes effectively undetectable after obfuscation. The collapse persists on repository-level tasks from SWE-bench Verified (Appendix M), and when AUROC is computed against four concrete negative classes instead of

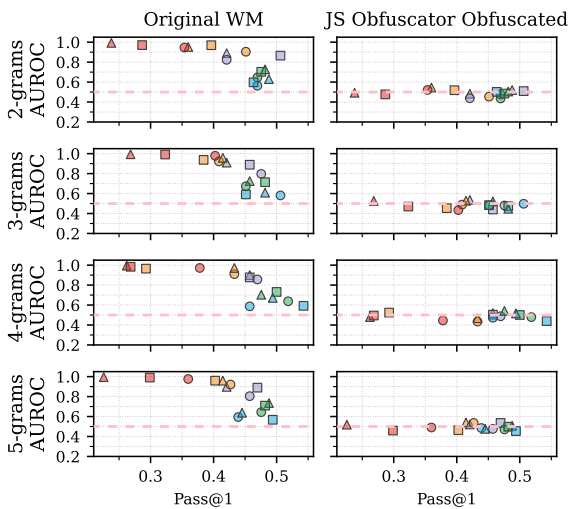


Figure 7: WLLM (Kirchenbauer et al., 2023) under 2- to 5-gram settings on LLaMA 3.1 (Grattafiori et al., 2024), before and after attack, on HumanEval-X-JS (Zheng et al., 2023). Left columns show original AUROC-Pass@1 trade-offs; right columns show attacked results.

the standard normal reference (Appendix N).

### 4.3 Impact of N-gram Length

As shown in Figure 7, we further test whether the attack depends on the chosen N-gram length. Using WLLM, JS Obfuscator, and LLaMA 3.1, we vary the N-gram length from 2 to 5. Before attack, WLLM is relatively insensitive to this choice: the average Pass@1 values are (0.419, 0.426, 0.429, 0.420), and the average AUROC values are (0.812, 0.818, 0.824, 0.819) for 2-through 5-grams. After the attack, our conclusion remains unchanged. All post-obfuscation AUROC scores cluster around 0.5, and no data point exceeds 0.6. This suggests that changing the N-gram order does not restore robustness.

### 4.4 Distribution Consistency Test

Finally, we directly test the core assumption behind our theory: distribution consistency (Assumption 3.1). On all three watermarking schemes from our experiments, we use a theory-grounded random-walk obfuscator (Appendix L) to sample within equivalent spaces. As seed programs, we take all highly watermarked programs (z-score  $> 4$ ) from our experiments that pass all HumanEval or MBPP test cases. This yields 1,620 Python programs with an average z-score of 5.9. For each seed program, we run 30 independent 100-step random walks; since the walk operates within the seed’s rule-based equivalent space, this constitutes random sampling within that space. We then apply the Anderson–Darling goodness-of-fit test (Stephens, 1974) to each equivalent space, with the null hypothesis  $H_0$  that the z-scores within the space follow  $\mathcal{N}(0, 1)$ , the standard normal distribution of the broader code space—that is, that distribution consistency holds for the space.

| Scheme  | Spaces | $H_0$ not rejected | Rate   |
|---------|--------|--------------------|--------|
| SWEET   | 867    | 827                | 95.39% |
| WLLM    | 700    | 678                | 96.86% |
| SynthID | 53     | 52                 | 98.11% |
| Overall | 1,620  | 1,557              | 96.11% |

Table 1: Anderson–Darling consistency test over 1,620 equivalent spaces sampled with the random-walk obfuscator. For each scheme, we report the number of equivalent spaces whose internal z-score distribution is not rejected as standard normal at the 5% significance level.

Table 1 summarizes the outcome. At the 5% significance level, the test does not reject  $H_0$  for 1,557

of the 1,620 equivalent spaces (96.11%), with per-scheme rates between 95.39% and 98.11%. Across all three watermarking schemes, a practical rule set can satisfy distribution consistency to a high degree, leaving little room for the watermark signal to survive within the equivalent space.

## 5 Conclusion

We show that N-gram-based code watermarking is fundamentally fragile under semantics-preserving obfuscation. Our theory models obfuscation as a Markov random walk and proves that, under the experimentally supported assumption of distribution consistency, robustness collapses: after enough obfuscation, detection fails at rate  $1 - \epsilon_{\text{pos}}$ .

Experiments across schemes, models, languages, benchmarks, and obfuscators confirm this prediction: detection degrades to near-random performance. This suggests that surface-form N-gram signals are insufficient for robust code watermarking, motivating semantics-aware designs.

## Limitations

**Attack on Other Watermarking Schemes** Recently, several watermarking schemes have been proposed that avoid the N-gram-based routine to improve robustness (Zhao et al., 2024; Liu and Bu, 2024), while this work is limited to N-gram-based schemes.

However, given that N-gram-based watermarking is both widely adopted (Dathathri et al., 2024; Kirchenbauer et al., 2023; Lee et al., 2024; Kirchenbauer et al., 2024; Liu et al., 2024; Wu et al., 2024; Zhao et al., 2025b; Fu et al., 2024) and industrially deployed (Dathathri et al., 2024), we believe that the robustness analysis of N-gram-based watermarking remains timely and valuable.

In addition, some studies inject watermarks using code transformation rules during post-processing or code LLM training dataset preparation (Yang et al., 2024a; Sun et al., 2023; Li et al., 2024). These approaches, as well as other software watermarking schemes (Dey et al., 2019), often rely on language-specific transformation rules. Although we focus on N-gram-based watermarking schemes, our attack would also disrupt such watermarks, since the transformation rules can be reversed directly by obfuscators, depending solely on whether the obfuscator implements them.

**Different Code Quality Aspects** This work evaluates code quality primarily through execution be-

havior, measured by whether generated programs pass test cases. Specifically, we use code benchmarks to compute the Pass@1 metric, following a widely adopted evaluation practice (Chen et al., 2021; Liu et al., 2023; Zheng et al., 2023; Cassano et al., 2023). We acknowledge that this does not cover all dimensions of code quality, such as readability, maintainability, or efficiency. Nevertheless, execution behavior is arguably the most fundamental dimension: its prominence in widely used code benchmarks reflects the fact that functional correctness is often the primary criterion for assessing generated code. In some settings, such as programming competitions, obfuscated binary deployment, and black-box API consumption, it may even be the main or only aspect of concern. In the future, developing obfuscation techniques that retain multiple dimensions of code quality could be a promising direction.

## Ethical Considerations

**Data sources and licensing.** Our experiments use public code-generation benchmarks and open-source obfuscation tools, including HumanEval, MBPP+, HumanEval-X-JS, MBPP-JS from MultiPL-E, and Python and JavaScript obfuscators. We use these resources only for research and evaluation, and we follow their original licenses and terms of use. Our released repository is intended to contain only our own code, evaluation scripts, and links or instructions for obtaining third-party resources from their original sources; we do not claim ownership over benchmark data, model outputs, or external software distributed by others.

**Privacy and sensitive information.** The datasets used in this work are benchmark programming tasks and tests rather than human-subject or user-interaction data, so we do not intentionally collect personal data or sensitive user information. We also do not deploy the studied watermarking or attack methods on real users, production systems, or private codebases.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 2342250, 2319944, and 2542053.

Research reported in this publication was supported by an Amazon Research Award, 2024, and an NVIDIA Academic Grant Program award, 2025.

We use AI writing assistance solely to improve the language of the manuscript, including paraphrasing or polishing the authors’ original text, rather than suggesting new content.

## References

- Scott Aaronson. 2023. [Watermarking of large language models](#). Accessed: 2024-12-12.
- Sebastian Banescu and Alexander Pretschner. 2018. A tutorial on software obfuscation. *Advances in Computers*, 108:283–353.
- Chandan Kumar Behera and D Lalitha Bhaskari. 2015. Different obfuscation techniques for code protection. *Procedia Computer Science*, 70:757–763.
- Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. [MultiPL-e: A scalable and polyglot approach to benchmarking neural code generation](#). *IEEE Trans. Softw. Eng.*, 49(7):3675–3691.
- Dominic Catacora. 2024. [Openai backs california bill to watermark ai-generated content](#). Allwork.Space. Accessed: 2026-05-17.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. [Codet: Code generation with generated tests](#). In *The Eleventh International Conference on Learning Representations*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. [Evaluating large language models trained on code](#). *CoRR*, abs/2107.03374.
- Minhao Cheng, Jinfeng Yi, Pin-Yu Chen, Huan Zhang, and Cho-Jui Hsieh. 2020. Seq2sick: Evaluating the robustness of sequence-to-sequence models with adversarial examples. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 3601–3608.
- Yong Cheng, Lu Jiang, and Wolfgang Macherey. 2019. [Robust neural machine translation with doubly adversarial inputs](#). In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4324–4333, Florence, Italy. Association for Computational Linguistics.
- Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubei, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. [Introducing SWE-bench verified](#).

- Jean-Pierre Corriveau, Vojislav Radonjic, and Wei Shi. 2014. Requirements verification: Legal challenges in compliance testing. In *2014 IEEE International Conference on Progress in Informatics and Computing*, pages 451–454. IEEE.
- Sumanth Dathathri, Abigail See, Sumedh Ghaisas, Po-Sen Huang, Robert Everett McAdam, Johannes Welbl, Vandana Bachani, Alex Kaskasoli, Robert Stanforth, Tatiana Matejovicova, Jamie Hayes, Nidhi Vyas, Majd Al Merey, Jonah Brown-Cohen, Rudy Bunel, Borja Balle, taylan. cemgil, Zahra Ahmed, Kitty Stacpoole, and 5 others. 2024. [Scalable watermarking for identifying large language model outputs](#). *Nature*, 634:818 – 823.
- Ayan Dey, Sukriti Bhattacharya, and Nabendu Chaki. 2019. Software watermarking: Progress and challenges. *INAE Letters*, 4:65–75.
- dflook. 2025. [Python-minifier](#). Accessed: 2026-05-19.
- Javid Ebrahimi, Daniel Lowd, and Dejing Dou. 2018. [On adversarial examples for character-level neural machine translation](#). In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 653–663, Santa Fe, New Mexico, USA. Association for Computational Linguistics.
- Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.
- Jiayi Fu, Xuandong Zhao, Ruihan Yang, Yuansen Zhang, Jiangjie Chen, and Yanghua Xiao. 2024. [Gumbel-Soft: Diversified language model watermarking via the GumbelMax-trick](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5791–5808, Bangkok, Thailand. Association for Computational Linguistics.
- GPTZero. [Ai detector - the original ai checker for chatgpt & more](#). Accessed: 2024-12-12.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. [The llama 3 herd of models](#). *Preprint*, arXiv:2407.21783.
- Batu Guan, Yao Wan, Zhangqian Bi, Zheng Wang, Hongyu Zhang, Pan Zhou, and Lichao Sun. 2024. [CodeIP: A grammar-guided multi-bit watermark for large language models of code](#). In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 9243–9258, Miami, Florida, USA. Association for Computational Linguistics.
- 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. 2024. [Deepseek-coder: When the large language model meets programming – the rise of code intelligence](#). *Preprint*, arXiv:2401.14196.
- Zhimeng Guo and Minhao Cheng. 2025. [Practical and effective code watermarking for large language models](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Venkatesan Guruswami. 2000. Rapidly mixing markov chains: A comparison of techniques. Available: [cs.washington.edu/homes/venkat/pubs/papers.html](http://cs.washington.edu/homes/venkat/pubs/papers.html).
- Chad Heitzenrater. 2024. [The case for and against ai watermarking](#). Federal Times. Accessed: 2026-05-17.
- Zhengmian Hu, Lichang Chen, Xidong Wu, Yihan Wu, Hongyang Zhang, and Heng Huang. 2024. [Unbiased watermark for large language models](#). In *The Twelfth International Conference on Learning Representations*.
- Mingjia Huo, Sai Ashish Somayajula, Youwei Liang, Ruisi Zhang, Farinaz Koushanfar, and Pengtao Xie. 2024. Token-specific watermarking with enhanced detectability and semantic coherence for large language models. In *Proceedings of the 41st International Conference on Machine Learning, ICML’24*. JMLR.org.
- Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. [MapCoder: Multi-agent code generation for competitive problem solving](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. [Live-codebench: Holistic and contamination free evaluation of large language models for code](#). In *The Thirteenth International Conference on Learning Representations*.
- javascript-obfuscator. 2026. [Javascript obfuscator](#). Accessed: 2026-05-19.
- Mark Jerrum. 2003. *Counting, sampling and integrating: algorithms and complexity*. Springer Science & Business Media.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. [SWE-bench: Can language models resolve real-world github issues?](#) In *The Twelfth International Conference on Learning Representations*.
- John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2023. A watermark for large language models. In *International Conference on Machine Learning*, pages 17061–17084. PMLR.

- John Kirchenbauer, Jonas Geiping, Yuxin Wen, Manli Shu, Khalid Saifullah, Kezhi Kong, Kasun Fernando, Aniruddha Saha, Micah Goldblum, and Tom Goldstein. 2024. [On the reliability of watermarks for large language models](#). In *The Twelfth International Conference on Learning Representations*.
- Stephen Cole Kleene. 1952. *Introduction to Metamathematics*. North-Holland, Groningen.
- Evgenios Konstantinou and Steffen Wolthusen. 2008. Metamorphic virus: Analysis and detection. *Royal Holloway University of London*, 15:15.
- Kalpesh Krishna, Yapei Chang, John Wieting, and Mohit Iyyer. 2022. Rankgen: Improving text generation with large ranking models. *arXiv preprint arXiv:2205.09726*.
- Kalpesh Krishna, Yixiao Song, Marzena Karpinska, John Wieting, and Mohit Iyyer. 2023. Paraphrasing evades detectors of ai-generated text, but retrieval is an effective defense. *Advances in Neural Information Processing Systems*, 36:27469–27500.
- Rohith Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. 2024. [Robust distortion-free watermarks for language models](#). *Transactions on Machine Learning Research*.
- Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. 2024. [Who wrote this code? watermarking for code generation](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4890–4911, Bangkok, Thailand. Association for Computational Linguistics.
- David A Levin and Yuval Peres. 2017. *Markov chains and mixing times*, volume 107. American Mathematical Soc.
- Boquan Li, Mengdi Zhang, Peixin Zhang, Jun Sun, and Xingmei Wang. 2024. Resilient watermarking for llm-generated codes. *arXiv preprint arXiv:2402.07518*, pages 1–18.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. [Competition-level code generation with alphacode](#). *Science*, 378(6624):1092–1097.
- liftoff. 2015. [Pyminifier](#). Accessed: 2026-05-19.
- Aiwei Liu, Leyi Pan, Xuming Hu, Shuang Li, Lijie Wen, Irwin King, and Philip S. Yu. 2024. [An unforgeable publicly verifiable watermark for large language models](#). In *The Twelfth International Conference on Learning Representations*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. [Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation](#). In *Thirty-seventh Conference on Neural Information Processing Systems*.
- Yepeng Liu and Yuheng Bu. 2024. Adaptive text watermark for large language models. *arXiv preprint arXiv:2401.13927*.
- Yan Luo. 2025. [China releases new labeling requirements for ai-generated content](#). Inside Privacy. Accessed: 2026-05-17.
- Elizabeth M Merkhofer, John Henderson, Abigail S Gertner, Michael Doyle, and Lily Wong. 2022. Practical attacks on machine translation using paraphrase. In *Proceedings of the 15th biennial conference of the Association for Machine Translation in the Americas (Volume 1: Research Track)*, pages 227–239.
- Meta. 2024. [Introducing llama 3.1: Our most capable models to date](#). Accessed: 2024-12-13.
- mishoo. 2024. [Uglifyjs](#). Accessed: 2026-05-19.
- Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In *International Conference on Machine Learning*, pages 24950–24962. PMLR.
- Kaiwen Ning, Jiachi Chen, Qingyuan Zhong, Tao Zhang, Yanlin Wang, Wei Li, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2024. Mcgmark: An encodable and robust online watermark for llm-generated malicious code. *arXiv preprint arXiv:2408.01354*.
- OpenAI, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, and 400 others. 2024. [Gpt-4o system card](#). *Preprint*, arXiv:2410.21276.
- Yanru Peng, Yuting Chen, and Beijun Shen. 2019. An adaptive approach to recommending obfuscation rules for java bytecode obfuscators. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 97–106. IEEE.
- Qwen. 2026. Qwen3.5-35b-a3b. <https://huggingface.co/Qwen/Qwen3.5-35B-A3B>.
- Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366.

- Vinu Sankar Sadasivan, Aounon Kumar, Sriram Balasubramanian, Wenxiao Wang, and Soheil Feizi. 2023. Can ai-generated text be reliably detected? *arXiv preprint arXiv:2303.11156*.
- Yash Shah, Jimil Shah, and Krishna Kansara. 2018. Code obfuscating a kotlin-based app with proguard. In *2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAEECC)*, pages 1–5. IEEE.
- Michael A Stephens. 1974. Edf statistics for goodness of fit and some comparisons. *Journal of the American statistical Association*, 69(347):730–737.
- Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. 2023. Codemark: Imperceptible watermarking for code datasets against neural code completion models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1561–1572.
- SWE-agent. 2025. mini-SWE-agent. <https://github.com/SWE-agent/mini-swe-agent>.
- Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. 2019. [Universal adversarial triggers for attacking and analyzing NLP](#). In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2153–2162, Hong Kong, China. Association for Computational Linguistics.
- Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.
- Wing Wong and Mark Stamp. 2006. Hunting for metamorphic engines. *Journal in Computer Virology*, 2:211–229.
- Scott Wu. 2024. [Introducing devin, the first ai software engineer](#). Accessed: 2024-12-11.
- Yihan Wu, Zhengmian Hu, Junfeng Guo, Hongyang Zhang, and Heng Huang. 2024. A resilient and accessible distribution-preserving watermark for large language models. In *Proceedings of the 41st International Conference on Machine Learning, ICML’24*. JMLR.org.
- Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024. Llm4fin: Fully automating llm-powered test case generation for fintech software acceptance testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1643–1655.
- Borui Yang, Wei Li, Liyao Xiang, and Bo Li. 2024a. Srcmarker: Dual-channel source code watermarking via scalable code transformations. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4088–4106. IEEE.
- John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.
- Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE.
- Hanlin Zhang, Benjamin L. Edelman, Danilo Francati, Daniele Venturi, Giuseppe Ateniese, and Boaz Barak. 2024. Watermarks in the sand: impossibility of strong watermarking for language models. In *Proceedings of the 41st International Conference on Machine Learning, ICML’24*. JMLR.org.
- Xinze Zhang, Junzhe Zhang, Zhenhua Chen, and Kun He. 2021. Crafting adversarial examples for neural machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1967–1977.
- Xuandong Zhao, Prabhanjan Vijendra Ananth, Lei Li, and Yu-Xiang Wang. 2024. [Provable robust watermarking for AI-generated text](#). In *The Twelfth International Conference on Learning Representations*.
- Xuandong Zhao, Sam Gunn, Miranda Christ, Jaiden Fairoze, Andres Fabrega, Nicholas Carlini, Sanjam Garg, Sanghyun Hong, Milad Nasr, Florian Tramèr, Somesh Jha, Lei Li, Yu-Xiang Wang, and Dawn Song. 2025a. [Sok: Watermarking for ai-generated content](#). In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2621–2639.
- Xuandong Zhao, Lei Li, and Yu-Xiang Wang. 2025b. [Permute-and-flip: An optimally stable and watermarkable decoder for LLMs](#). In *The Thirteenth International Conference on Learning Representations*.
- Zhengli Zhao, Dheeru Dua, and Sameer Singh. 2018. Generating natural adversarial examples. In *International Conference on Learning Representations (ICLR)*.
- 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. 2023. [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*, page 5673–5684, New York, NY, USA. Association for Computing Machinery.

Tong Zhou, Xuandong Zhao, Xiaolin Xu, and Shaolei Ren. 2024. [Bileve: Securing text provenance in large language models against spoofing with bi-level signature](#). In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

## A N-gram-based Watermark

In the discourse on tracing and regulating AI-generated content (AIGC), watermarking offers a promising solution. Among various approaches, those that modify the output probability distribution of a language model during generation are of particular interest due to their model-agnostic nature. Following the foundational framework proposed by (Kirchenbauer et al., 2023), which embeds a signal by modifying vocabulary logits, we provide a detailed definition and workflow of the N-gram-based watermarking technique. This process is divided into two core stages: **Watermark Embedding** and **Watermark Detection**.

### A.1 Watermark Embedding

Watermark embedding occurs at every token generation step of the Large Language Model (LLM). The objective is to slightly alter the sampling probability distribution of the next token to favor a verifiable pattern, without significantly degrading the quality of the text.

#### Context-based Pseudorandom Seed Generation.

During the generation of the  $i$ -th token,  $t_i$ , the watermarking algorithm uses the preceding  $N - 1$  tokens ( $t_{i-N+1}, \dots, t_{i-1}$ ) as context, so that the context together with the current token spans an N-gram. This context is used to generate a deterministic, pseudorandom seed.

A seed  $s_i$  is generated using a public hash function  $H$  (e.g., SHA-256):

$$s_i = H(t_{i-N+1}, \dots, t_{i-1}).$$

To enhance security, a secret key  $k$  known only to the owner can be incorporated:  $s_i = H(t_{i-N+1}, \dots, t_{i-1}, k)$ . In the simplest case,  $N = 2$ , the context is the single preceding token  $t_{i-1}$ . The first  $N - 1$  tokens of a generation are sampled from the unmodified distribution, since no full  $(N - 1)$ -token context is available for them.

**Vocabulary Partitioning.** After obtaining the seed  $s_i$ , it is used to initialize a Pseudorandom Number Generator (PRNG), which partitions the entire vocabulary  $V$  into two disjoint sets in a reproducible manner: **Green List** ( $G_i$ ) and **Red List** ( $R_i$ ).

The partition is governed by a hyperparameter  $\gamma \in (0, 1)$ , which represents the fraction of the vocabulary assigned to the green list.

$$V = G_i \cup R_i \quad \text{and} \quad G_i \cap R_i = \emptyset,$$

$$|G_i| \approx \gamma|V|.$$

Since  $s_i$  is determined by the context ( $t_{i-N+1}, \dots, t_{i-1}$ ), the contents of the green list  $G_i$  and red list  $R_i$  are identical for the same context.

**Logit Modification.** After the LLM generates the original logits vector  $l_i = (l_{i,1}, l_{i,2}, \dots, l_{i,|V|})$  for position  $i$ , the watermarking algorithm modifies it. Specifically, it adds a fixed bias  $\delta > 0$  to the logits of all tokens in the green list  $G_i$ .

The modified logits vector  $l'_i$  is computed as follows:

$$l'_{i,v} = \begin{cases} l_{i,v} + \delta & \text{if token } v \in G_i, \\ l_{i,v} & \text{if token } v \in R_i. \end{cases}$$

This operation significantly increases the probability of sampling tokens from the green list. Finally, the model samples the  $i$ -th token  $t_i$  from this modified probability distribution:

$$t_i \sim \text{Softmax}(l'_i).$$

### A.2 Watermark Detection

The detection process aims to determine if a given text  $T = (t_1, t_2, \dots, t_L)$  contains a watermark.

**Reconstructing the Green Lists.** The detector possesses the same hash function  $H$  (and secret key  $k$ ) used during embedding. For each token  $t_i$  in the text (from  $i = N$  to  $L$ ), the detector uses the preceding  $N - 1$  tokens ( $t_{i-N+1}, \dots, t_{i-1}$ ) to regenerate the exact green list  $G_i$  that was used at the time of generation. The first  $N - 1$  tokens, which lack a full context, are not scored.

**Hypothesis Testing.** The detection process is a standard statistical hypothesis test:

- **Null Hypothesis ( $H_0$ ):** The text was not generated by a watermarked LLM. Therefore, the probability of any token  $t_i$  appearing in its corresponding green list  $G_i$  is approximately equal to the green list proportion,  $\gamma$ .
- **Alternative Hypothesis ( $H_A$ ):** The text was generated by a watermarked LLM. Due to the logit modification, the probability of a token  $t_i$  appearing in its green list  $G_i$  is significantly greater than  $\gamma$ .

**Calculating the Test Statistic.** First, among the  $m = L - N + 1$  scored positions, we count the total number of tokens  $K$  that fall into their corresponding green lists:

$$K = \sum_{i=N}^L \mathbb{1}(t_i \in G_i).$$

where  $\mathbb{1}(\cdot)$  is the indicator function (1 if the condition is true, 0 otherwise).

Under the null hypothesis  $H_0$ , the count  $K$  should follow a binomial distribution  $B(m, \gamma)$ . For a sufficiently long text (large  $m$ ), this binomial distribution can be approximated by a normal distribution.

We can then compute a **z-score** to measure how much the observed count  $K$  deviates from the expected count:

$$z = \frac{K - \mu}{\sigma} = \frac{K - m\gamma}{\sqrt{m\gamma(1 - \gamma)}}.$$

where:

- $K$  is the observed count of green-list tokens.
- $\mu = m\gamma$  is the expected value of  $K$  under  $H_0$ .
- $\sigma = \sqrt{m\gamma(1 - \gamma)}$  is the standard deviation of  $K$  under  $H_0$ .

A sufficiently large positive z-score (e.g.,  $z > 4$  or 5) provides strong statistical evidence to reject the null hypothesis  $H_0$ , leading to the conclusion that the text is watermarked.

## B Notion Formalizations

In this section, we formalize the main concepts used in our theory, partly following prior definitions in (Zhang et al., 2024; Zhao et al., 2025a).

### B.1 Threat Model

Let  $\mathcal{X}$  be a prompt space,  $\mathcal{C}$  a code space,  $M : \mathcal{X} \rightarrow \mathcal{C}$  a randomized code generation model, and  $T : \mathcal{X} \times \mathcal{C} \rightarrow \{0, 1\}$  a code test suite that checks whether a candidate program satisfies a prompt. A secret-key watermarking scheme  $\Pi = (\text{Watermark}, \text{Detect})$  consists of two algorithms:

- **Watermark( $M$ ):** Given  $M \in \mathcal{M}$ , this randomized algorithm outputs a secret key  $k \in \mathcal{K}$  and a watermarked model  $M_k : \mathcal{X} \rightarrow \mathcal{C}$  parameterized by  $k$ .

- **Detect $_k(x, c)$ :** Given a secret key  $k \in \mathcal{K}$ , a prompt  $x \in \mathcal{X}$ , and a code sample  $c \in \mathcal{C}$ , this deterministic detector outputs  $\text{Detect}_k(x, c) \in \{0, 1\}$ , where 1 indicates that the watermark is present and 0 indicates that it is absent.

The goal is that  $M_k$  generates watermarked code that can later be recognized by  $\text{Detect}_k$ . The secret key  $k$  may be chosen and maintained by the model provider or by another designated party.

We evaluate the detector using the false negative rate (FNR)

$$\epsilon_{\text{neg}} \leftarrow \mathbb{E}(\mathbb{1}[\text{Detect}_k(x, c) = 0] \mid c \in \mathcal{C}_k)$$

and the false positive rate (FPR)

$$\epsilon_{\text{pos}} \leftarrow \mathbb{E}(\mathbb{1}[\text{Detect}_k(x, c) = 1] \mid c \in \mathcal{C}),$$

where  $\mathcal{C}_k \subseteq \mathcal{C}$  denotes the set of watermarked responses produced by  $M_k$ . For a usable watermarking scheme, both  $\epsilon_{\text{pos}}$  and  $\epsilon_{\text{neg}}$  should be sufficiently small.

For a prompt  $x$ , let  $\mathcal{F}$  denote the behavior requested by the prompt, and let  $\Phi^{\mathcal{F}} \subseteq \mathcal{C}$  denote the *high-quality space*, namely the set of programs  $c$  such that  $T(x, c) = 1$ . Code quality can be measured along many dimensions, but in this work both  $\mathcal{F}$  and  $T$  refer only to runtime behavior, such as correctness and functional compliance. This focus follows standard practice in code-generation benchmarks, where execution behavior is the primary quality criterion (Chen et al., 2021; Liu et al., 2023; Zheng et al., 2023; Cassano et al., 2023).

**Adversary Attacker:** An adversary  $\text{Att} : \mathcal{X} \times \mathcal{C} \rightarrow \mathcal{C}$  acts as a code modifier or obfuscator and satisfies

$$\forall c \in \Phi^{\mathcal{F}}, \text{Att}(x, c) \in \Phi^{\mathcal{F}}.$$

Equivalently,  $T(x, \text{Att}(x, c)) = 1$  whenever  $T(x, c) = 1$ . Thus, given a prompt  $x$  and a high-quality program  $c$ , the attacker outputs a transformed program  $c' \leftarrow \text{Att}(x, c)$  without degrading execution quality.

We use  $\epsilon$ -breaking to formalize the attack goal. An attacker  $\epsilon$ -breaks the watermarking scheme if, for every  $M \in \mathcal{M}$  and every prompt  $x \in \mathcal{X}$ , we have

$$\mathbb{E}(\mathbb{1}[\text{Detect}_k(x, \text{Att}(x, c)) = 0] \mid c \in \mathcal{C}_k) \geq \epsilon,$$

that is, after the quality-preserving transformation by  $\text{Att}$ , the resulting false negative rate is at least  $\epsilon$ .

From the defender’s perspective, a watermarking scheme is robust if every feasible attacker can only force  $\epsilon$  to remain close to the baseline  $\epsilon_{\text{neg}}$ . In that case, the detector continues to work even after adversarial code modifications.

## C Equivalent Transformations

In this section, we formalize equivalent transformation rules and the rule-based equivalent spaces they induce. The general idea of changing program syntax while preserving behavior is standard in compiler optimization, refactoring, and code obfuscation (Behera and Bhaskari, 2015; Banescu and Pretschner, 2018; Peng et al., 2019; Shah et al., 2018). Our attack model builds on this idea: it treats semantics-preserving transformations as the moves of a random walk over high-quality programs.

**Definition C.1 (Equivalent Transformation Rule and Executor).** Given a program  $c \in \mathcal{C}$ , we use two algorithms to model equivalent transformations:

- $R(c)$ : Each rule  $R$  is an action-derivation algorithm. Given  $c$ , it returns a set of executable actions,

$$R : \mathcal{C} \rightarrow \bigcup_{i=1}^{\infty} \mathcal{A}^i,$$

that can be applied to  $c$  by  $\text{Transform}(\cdot, \cdot)$  below. Different rules in the rule space  $\mathcal{R}$  correspond to different transformation types. Here  $a \in \mathcal{A}$  denotes one concrete transformation action.

- $\text{Transform}(c, a)$ : This executor is shared across rules. Given a program and an action  $a \in \mathcal{A}$  compatible with the current rule,  $\text{Transform} : \mathcal{C} \times \mathcal{A} \rightarrow \mathcal{C}$  performs a quality-preserving transformation satisfying

$$\forall \mathcal{F}, \forall c \in \Phi^{\mathcal{F}}, \text{Transform}(c, a) \in \Phi^{\mathcal{F}},$$

where  $a$  must be one of the actions allowed by the rule currently under consideration.

The point of this abstraction is to separate *what kind of transformation* is allowed from *which concrete modification* is executed. A rule describes a transformation family, whereas  $\text{Transform}$  applies one specific action from that family.

For example, one rule may correspond to “delete a comment,” in which each returned action deletes

---

### Algorithm 1: RandomStep

---

**Input:** Original Code:  $c_{\text{ori}} \in \mathcal{C}$ ,  
Rule Set:  $\Gamma = \{R_1, R_2, \dots, R_n\}$

**Output:** Transformed Code:  $c_{\text{trans}} \in \mathcal{C}$

```

1 Init Action Set  $\Lambda \leftarrow \emptyset$ 
2 for each  $i$  in  $1, 2, \dots, n$  do
3    $\Lambda \leftarrow \Lambda \cup R_i(c_{\text{ori}})$ 
4 Init  $j \leftarrow \text{RandomChoose}(\{1, 2, \dots, |\Lambda|\})$ 
   //  $\Lambda$  is  $\{a_1, a_2, \dots, a_{|\Lambda|}\}$ , and  $\Lambda \subseteq \mathcal{A}$ 
5 Init  $c_{\text{trans}} \leftarrow \text{Transform}(c_{\text{ori}}, a_j)$ 
6 Return  $c_{\text{trans}}$ 

```

---

one specific comment at one specific location in  $c$ . Another rule may correspond to “rewrite a for-loop as a while-loop,” in which each action targets one concrete loop instance. Many such quality-preserving rules are readily implementable in practice, including comment removal, variable renaming, and dead-code insertion (dflook, 2025; liftoff, 2015; javascript-obfuscator, 2026; mishoo, 2024).

We now define the randomized transformation procedure used in our attack.

**Definition C.2 (Random Transformation).** Let  $\text{RandomChoose} : \bigcup_{i=1}^{\infty} \mathbb{N}^i \rightarrow \mathbb{N}$  be an algorithm that samples one integer uniformly from a finite integer set. Using  $\text{RandomChoose}$ , we define two randomized algorithms,  $\text{RandomStep}(c, \Gamma)$  and  $\text{RandomWalk}(c, \Gamma, t)$ , in Algorithms 1 and 2.

The algorithm  $\text{RandomStep}$  (Algorithm 1) performs one quality-preserving transformation on the current program. It first aggregates all actions enabled by the rule set  $\Gamma$  and then selects one action uniformly at random from the resulting set  $\Lambda$  (line 4). The algorithm  $\text{RandomWalk}$  applies  $\text{RandomStep}(c, \Gamma)$  repeatedly for a finite number of steps  $t$ .

Importantly, in our model, neither  $\text{RandomStep}$  nor  $\text{RandomWalk}$  depends on the prompt or on prompt-specific semantic annotations. In software engineering practice, many semantics-preserving transformations can be carried out directly from the program text once the language grammar is known.

We next define the two central notions used in the analysis: ergodicity rule sets and rule-based equivalent spaces.

**Definition C.3 (Ergodicity Rule Set).** An ergodicity rule set  $\Gamma_{\text{ERG}} = \{R_1, R_2, \dots, R_n\}$  is a set of rules satisfying the following properties:

- **Empty Rule:** The set contains a rule that per-

forms empty transformations. That is, for all  $x \in \mathcal{X}$ ,  $c \in \mathcal{C}$ , and  $a \in R(c)$ , the corresponding executor satisfies

$$c = \text{Transform}(c, a).$$

- **Inverse Rule:** Every rule  $R \in \Gamma_{\text{ERG}}$  has an inverse rule  $R^{-1}$  such that, if  $\Lambda = R(c)$  and  $c' = \text{Transform}(c, a)$ , then for each action  $a \in \Lambda$  there exists at least one inverse action  $a^{-1} \in R^{-1}(c')$  satisfying

$$c = \text{Transform}(c', a^{-1}).$$

Consequently, every multi-step transformation is reversible in the reachability sense. For brevity, write

$$p_t(c_i, c_j) \quad (1)$$

$$:= \quad (2)$$

$$\Pr[c_j = \text{RandomWalk}(c_i, \Gamma_{\text{ERG}}, t)]. \quad (3)$$

Then, for all  $c_+, c_- \in \mathcal{C}$  and all  $t \in \mathbb{N}$ ,

$$p_t(c_+, c_-) > 0 \iff p_t(c_-, c_+) > 0.$$

An ergodicity rule set is not merely a theoretical construct; it can be implemented using standard code transformations. Table 2 gives one example in which each rule has a corresponding inverse rule. Some rules can even be self-inverse. For instance, although reversing comment deletion by randomly adding a comment may have low probability, that probability is still nonzero, which is sufficient for Definition C.3.

#### Definition C.4 (Rule-based Equivalent Space).

Given a prompt  $x$ , an initial code sample  $c \in \Phi^{\mathcal{F}}$ , a rule set  $\Gamma = \{R_1, \dots, R_n\}$ , and an arbitrary finite step count  $t \in \mathbb{N}$ , the rule-based equivalent space  $\Phi_c^{\Gamma}$  consists of the initial code  $c$  and all programs reachable as  $\text{RandomWalk}(c, \Gamma, t)$ . In particular,  $\Phi_c^{\Gamma} \subseteq \Phi^{\mathcal{F}} \subseteq \mathcal{C}$ .

Equivalently,  $\Phi_c^{\Gamma}$  collects every high-quality program that can be reached from  $c$  by applying the transformations in  $\Gamma$  for any finite number of steps.

| R   | $R^{-1}$ | DESCRIPTION                     |
|-----|----------|---------------------------------|
| I   | I        | Do nothing.                     |
| II  | II       | Randomly rename variables.      |
| III | IV       | Add a random comment.           |
| IV  | III      | Delete a comment.               |
| V   | VI       | Add a random dead code snippet. |
| VI  | V        | Delete a dead code snippet.     |

Table 2: Example of an ergodicity rule set. The left column lists  $R$  and the middle column lists  $R^{-1}$ .

---

#### Algorithm 2: RandomWalk

---

**Input:** Original Code:  $c_{\text{ori}} \in \mathcal{C}$ ,  
Rule Set:  $\Gamma = \{R_1, R_2, \dots, R_n\}$ ,  
Number of Steps:  $t \in \mathbb{N}$

**Output:** Transformed Code:  $c_{\text{trans}} \in \mathcal{C}$

---

```

1 Init  $c \leftarrow c_{\text{ori}}$ 
2 for each  $i$  in  $1, 2, \dots, t$  do
3    $\lfloor$  Set  $c \leftarrow \text{RandomStep}(c, \Gamma)$ 
4 Init  $c_{\text{trans}} \leftarrow c$ 
5 Return  $c_{\text{trans}}$ 

```

---

### C.1 Chain Modeling

We now model rule-based code transformation as a graph, because our attacker operates as the random walk defined in Definition C.2.

Given an initial program  $c \in \mathcal{C}$  and a rule set  $\Gamma$ , the induced rule-based equivalent space  $\Phi_c^{\Gamma}$  gives rise to a weighted directed graph  $G_c^{\Gamma} = (V, E, W)$  describing the possible transformations inside that space. If  $\Phi_c^{\Gamma} = \{c_1, c_2, \dots, c_n\}$ , then each vertex  $v_i \in V$  corresponds to program  $c_i$ . An edge  $e_{ij} \in E$  exists whenever

$$\Pr[c_j = \text{RandomStep}(c_i, \Gamma)] > 0,$$

meaning that  $c_i$  can be transformed into  $c_j$  in one step. Each such edge carries weight  $w_{ij} = \Pr[c_j = \text{RandomStep}(c_i, \Gamma)] \in (0, 1]$ .

Because Algorithm 1 chooses uniformly from the available action set, all outgoing edges from the same vertex have equal weight. Formally,

$$\forall w_{i+}, w_{i-} \in \{w_{ij} \mid e_{ij} \in E, \forall j\}, \quad w_{i+} = w_{i-}.$$

Based on  $G_c^{\Gamma}$ , we define the transition matrix  $\vec{P} \in \mathbb{R}^{|V| \times |V|}$  by

$$\vec{P}_{ij} = \begin{cases} w_{ij}, & e_{ij} \in E, \\ 0, & \text{otherwise.} \end{cases}$$

For each starting program  $c_i$ , define the probability distribution vector  $p(c_i, t) \in \mathbb{R}^{|V|}$  by

$$p(c_i, t)_j = \Pr[c_j = \text{RandomWalk}(c_i, \Gamma, t)].$$

Thus,  $p(c_i, t)_j \in [0, 1]$  is the probability that the random walk, started from  $c_i$ , is at  $c_j$  after  $t$  calls to  $\text{RandomStep}$  (line 3). For example,  $p(c_1, 0) = [1, 0, 0, \dots, 0]$  represents the initial state concentrated on  $c_1$ . By construction,

$$p(c, t+1) = \vec{P}^{\top} \cdot p(c, t).$$

If  $\vec{P}$  satisfies the standard conditions for the existence of a stationary distribution, namely irreducibility and aperiodicity of the corresponding Markov chain (Levin and Peres, 2017), then there exists a stationary distribution  $\pi \in \mathbb{R}^{|V|}$  such that

$$\exists \pi, \forall c \in \Phi_c^\Gamma, \lim_{t \rightarrow \infty} p(c, t) = \pi.$$

Following classical Markov-chain theory, we further define the mixing time

$$t_i(\epsilon) = \min \{t \mid \|p(c_i, t) - \pi\|_{\text{TV}} \leq \epsilon\},$$

which denotes the number of steps required for the walk started from  $c_i$  to come within total variation distance  $\epsilon$  of the stationary distribution  $\pi$ .

## D Model Analysis

In this section, we analyze the structure induced by ergodicity rule sets. We first show in Appendix D.1 that the corresponding code-transformation graph admits a stationary distribution. Then, in Appendix D.2, we prove that an ergodicity rule set induces a partition of the high-quality code space.

### D.1 The Stationary Distribution Exists

We now show that, inside the rule-based equivalent space induced by an ergodicity rule set, the random walk RandomWalk has a stationary distribution that does not depend on the starting program. By standard Markov-chain theory (Levin and Peres, 2017), it suffices to prove irreducibility and aperiodicity.

**Theorem D.1 (Inside-Space Independence).** Let  $\Phi_c^{\Gamma_{\text{ERG}}}$  be a rule-based equivalent space generated by an ergodicity rule set (Definition C.3). Then

$$\exists \pi, \forall c' \in \Phi_c^{\Gamma_{\text{ERG}}}, \lim_{t \rightarrow \infty} p(c', t) = \pi,$$

where  $\Phi_c^{\Gamma_{\text{ERG}}}$  is defined in Appendix C.4 and  $p(\cdot)$  is defined in Appendix C.1. Equivalently, the associated code-transformation graph  $G_c^{\Gamma_{\text{ERG}}}$  is irreducible and aperiodic.

*Proof of Theorem D.1's Irreducibility.* By Definition C.4, every program in  $\Phi_c^{\Gamma_{\text{ERG}}}$  is reachable from the original program  $c$ . Hence,

$$\forall c' \in \Phi_c^{\Gamma_{\text{ERG}}}, \exists t, \Pr[c \rightarrow c' \mid t] > 0,$$

where  $\Pr[c \rightarrow c' \mid t]$  abbreviates  $\Pr[c' = \text{RandomWalk}(c, \Gamma_{\text{ERG}}, t)]$ .

Moreover, by the inverse-rule property in Definition C.3, for all  $c_+, c_- \in \mathcal{C}$ ,

$$\begin{aligned} \Pr[c_+ \rightarrow c_- \mid t] > 0 &\iff \\ \Pr[c_- \rightarrow c_+ \mid t] > 0. \end{aligned}$$

Now fix arbitrary  $c_+, c_- \in \Phi_c^{\Gamma_{\text{ERG}}}$ . Since both are reachable from  $c$ , there exist  $t_+, t_- \in \mathbb{N}$  such that

$$\Pr[c \rightarrow c_+ \mid t_+] > 0, \quad \Pr[c \rightarrow c_- \mid t_-] > 0.$$

By reversibility of reachability, we also have

$$\Pr[c_+ \rightarrow c \mid t_+] > 0.$$

Define

$$\begin{aligned} p_+ &:= \Pr[c_+ \rightarrow c \mid t_+], \\ p_- &:= \Pr[c \rightarrow c_- \mid t_-]. \end{aligned}$$

Then

$$\Pr[c_+ \rightarrow c_- \mid t_+ + t_-] \geq p_+ p_- > 0.$$

Therefore, every pair of vertices in  $G_c^{\Gamma_{\text{ERG}}}$  is mutually reachable through a finite path passing through  $c$ . The graph is thus irreducible.  $\square$

*Proof of Theorem D.1's Aperiodicity.* By the empty-rule property in Definition C.3, the graph contains a self-loop at every program with positive probability. In particular,

$$\Pr[c = \text{RandomStep}(c, \Gamma_{\text{ERG}})] > 0.$$

Let

$$p_{\text{self}} := \Pr[c = \text{RandomStep}(c, \Gamma_{\text{ERG}})].$$

Hence, for every  $t \in \mathbb{N}$ ,

$$\Pr[c = \text{RandomWalk}(c, \Gamma_{\text{ERG}}, t)] \geq p_{\text{self}}^t > 0.$$

Therefore, every positive integer belongs to the return-time set,

$$\{t \mid c = \text{RandomWalk}(c, \Gamma, t)\} = \mathbb{N},$$

and thus its greatest common divisor is 1. The graph is aperiodic.  $\square$

Note that  $\Phi_c^{\Gamma_{\text{ERG}}}$  depends on both the rule set  $\Gamma_{\text{ERG}}$  and the initial program  $c$ . Once these two objects are fixed, the corresponding space is uniquely determined.

Thus, Theorem D.1 states that within a fixed equivalent space, the long-run behavior of the walk is intrinsic to the space itself: regardless of the starting program, the distribution converges to the same stationary distribution  $\pi$ .

## D.2 Rule-based Partition

For the later impossibility result, we next show that an ergodicity rule set partitions the high-quality code space into disjoint equivalent spaces.

**Theorem D.2 (Ergodicity Rule Set can Partition the High-Quality Code Space).** Given an ergodicity rule set  $\Gamma_{\text{ERG}}$ , for all  $c_+, c_- \in \Phi^{\mathcal{F}}$  we have

$$\neg(\Phi_{c_+}^{\Gamma_{\text{ERG}}} = \Phi_{c_-}^{\Gamma_{\text{ERG}}}) \iff \Phi_{c_+}^{\Gamma_{\text{ERG}}} \cap \Phi_{c_-}^{\Gamma_{\text{ERG}}} = \emptyset$$

where  $\Phi^{\mathcal{F}} \subseteq \mathcal{C}$ . In other words, two distinct rule-based equivalent spaces do not overlap, so  $\Gamma_{\text{ERG}}$  induces a partition of the high-quality code space.

*Proof of Theorem D.2.* We abbreviate  $\Phi_{c_+}^{\Gamma_{\text{ERG}}}$  and  $\Phi_{c_-}^{\Gamma_{\text{ERG}}}$  as  $\Phi_+$  and  $\Phi_-$ , and reuse the notation  $\Pr[\rightarrow]$  from the proof of Theorem D.1. Assume

$$\Phi := \Phi_+ \cap \Phi_- \neq \emptyset.$$

Let  $c \in \Phi$ . Then for every  $c_+ \in \Phi_+$ , there exists  $t_+ \in \mathbb{N}$  such that

$$\Pr[c_+ \rightarrow c \mid t_+] > 0.$$

Likewise, for every  $c_- \in \Phi_-$ , there exists  $t_- \in \mathbb{N}$  such that

$$\Pr[c \rightarrow c_- \mid t_-] > 0.$$

Define

$$\begin{aligned} p_+ &:= \Pr[c_+ \rightarrow c \mid t_+], \\ p_- &:= \Pr[c \rightarrow c_- \mid t_-]. \end{aligned}$$

Then, for all  $c_+ \in \Phi_+$  and  $c_- \in \Phi_-$ ,

$$\Pr[c_+ \rightarrow c_- \mid t_+ + t_-] \geq p_+ p_- > 0.$$

Hence every  $c_- \in \Phi_-$  is reachable from every  $c_+ \in \Phi_+$ , so  $\Phi_- \subseteq \Phi_+$ . By symmetry, we also obtain  $\Phi_+ \subseteq \Phi_-$ . Therefore,

$$\Phi_+ = \Phi_-.$$

□

**Definition D.1 (Rule-based Partition).** Following Theorem D.2, given a high-quality code space  $\Phi^{\mathcal{F}}$  and an ergodicity rule set  $\Gamma_{\text{ERG}}$ , the induced rule-based partition  $\mathcal{P}^{\Gamma_{\text{ERG}}}$  is

$$\mathcal{P}^{\Gamma_{\text{ERG}}} = \left\{ \Phi_1^{\Gamma_{\text{ERG}}}, \Phi_2^{\Gamma_{\text{ERG}}}, \dots, \Phi_n^{\Gamma_{\text{ERG}}} \right\}$$

and it satisfies the following properties (abbreviating  $\mathcal{P}^{\Gamma_{\text{ERG}}}$  as  $\mathcal{P}$ ):

$$\begin{aligned} \forall \Phi \in \mathcal{P}, \Phi &\subseteq \Phi^{\mathcal{F}}; \\ \bigcup_{\Phi \in \mathcal{P}} \Phi &= \Phi^{\mathcal{F}}; \\ \forall \Phi_+, \Phi_- \in \mathcal{P}, \Phi_+ \cap \Phi_- &= \emptyset. \end{aligned}$$

Thus,  $\mathcal{P}^{\Gamma_{\text{ERG}}}$  is precisely the partition of  $\Phi^{\mathcal{F}}$  induced by  $\Gamma_{\text{ERG}}$ .

## E Impossibility Results

### E.1 Assumption and Robustness Impossibility

In this section, we will describe our only assumption, which is intuitive and has been supported by experiments.

First, we define the target of our assumption, distribution consistency.

**Definition E.1 (Distribution Consistency).** We define  $\mathcal{Q}(\mathcal{C})$  to denote a distribution over the code space  $\mathcal{C}$ , where  $\mathcal{Q}$  is independent of the watermarking scheme. Given a partition  $\mathcal{P}^{\Gamma_{\text{ERG}}}$ , a watermarking scheme  $\Pi = (\text{Watermark}, \text{Detect})$  satisfies the *distribution consistency* property if, for all  $\mathcal{Q}$ ,  $x \in \mathcal{X}$ ,  $c \in \mathcal{C}$ , and  $k \in \mathcal{K}$ , given  $c \sim \mathcal{Q}(\mathcal{C})$ ,  $\text{Detect}_k(x, c) \sim \mathcal{D}$ , we have:

$$\text{if } c \sim \mathcal{Q}(\Phi), \text{ then } \text{Detect}_k(x, c) \sim \mathcal{D}.$$

That is, the distribution of  $\text{Detect}_k$  within each equivalent space  $\Phi \in \mathcal{P}^{\Gamma_{\text{ERG}}}$  remains the same as the distribution observed over the entire code space.

Recall that distribution consistency is a necessary and sufficient condition for the independence between  $\Gamma_{\text{ERG}}$  and  $\text{Detect}$ . Moreover, the  $\text{Detect}(x, c)$  among the whole code space  $\mathcal{C}$  follows the Bernoulli distribution,  $\text{Bernoulli}(\epsilon_{\text{pos}})$ .

**Assumption E.1 (Impossibility Assumption).** With implementable code transformation rules, distribution consistency (Definition E.1) can be achieved for N-gram-based watermarking schemes.

**Discussion of Assumption E.1:** First, distribution consistency is clearly not guaranteed for arbitrary rule sets. For example, if a rule set only makes minor surface changes, many literal features may be preserved, and the resulting samples can remain biased under N-gram-based watermark detectors. An extreme case is a rule set containing only the identity rule, which produces no perturbation at all.

Nevertheless, the assumption becomes more plausible for sufficiently expressive transformation rule sets. Modern syntax- and semantics-preserving transformations can substantially vary the N-gram features within an equivalent space. For example, variable names, string or numeric constants, local control/data flow, and expressions can all be perturbed to different degrees, often independently of the N-gram-level features used by watermark detectors. By contrast, the components that tend to remain stable under such transformations are usually semantic in nature, such as design patterns, high-level control/data flow, and functionality. Since N-gram-based watermarking schemes are not aware of these semantic features, it is reasonable to assume that their detectable N-gram features are largely independent of the equivalent-space partition.

We can think of distribution consistency as the product of a game between watermarking schemes and attackers. If the watermarking scheme can watermark and detect code properties at a higher level, e.g., syntax or even functional level, while the rule set can only union codes with low-level equivalence, the distribution inside rule-based equivalent space will be seriously biased. For example, if the watermarking scheme can watermark and detect the AST pattern, while the rule set can only disturb variable names, the Detect will return the same results inside each equivalent space. On the other side, if the watermarking scheme can only watermark and detect code properties at a low level, like the N-gram level, the distribution consistency is easier to satisfy by existing syntax or semantic level transformation rules.

Based on the discussion above, we make Assumption E.1 and empirically test it in Subsection 4.4.

**Theorem E.1 (Impossibility Theorem).** With Assumption 3.1 satisfied, given large enough  $t$ , there exists an ergodicity rule set  $\Gamma_{\text{ERG}}$  that for all N-gram-based schemes with false positive rate  $\epsilon_{\text{pos}}$ , the  $\text{RandomWalk}(c \mid c \in \mathcal{C}_k, \Gamma_{\text{ERG}}, t)$  algorithm can perform as an attacker and  $(1 - \epsilon_{\text{pos}})$ -break the watermarking.

*Proof of Theorem E.1.* We remind that:

- The secret key  $k$  is sampled randomly and independently of  $(x, c) \in \mathcal{X} \times \mathcal{C}$ , which ensures that we can bound the false positive rate without needing to make assumptions on the

unknown human-generated data distribution.

- For each ergodicity rule set, given long enough  $t$ , start from a watermarked code  $c$ , after  $\text{RandomWalk}(c, \Gamma_{\text{ERG}}, t)$  runs, the results (and the stationary distribution  $\pi$ ) only rely on: **1)** Which equivalent spaces  $\Phi_c^{\Gamma_{\text{ERG}}}$  the original  $c$  belongs to. **2)** The stationary distribution  $\pi$  of  $G_c^{\Gamma_{\text{ERG}}}$ .
- We assume the distribution consistency (Assumption 3.1), i.e., the independence between  $\Gamma_{\text{ERG}}$  and Detect.

Therefore, if we set:

- $\text{Att}(x, c) := \text{RandomWalk}(c, \Gamma_{\text{ERG}}, t)$  (the prompt  $x$  is provided as an input here, but note that the algorithm  $\text{RandomWalk}$  does not rely on it),
- For  $\mathcal{P}^{\Gamma_{\text{ERG}}} = \{\Phi_1, \Phi_2, \dots, \Phi_m\}$ , the probability distribution  $r \in \mathbb{R}^m$  has  $r_j = \mathbb{E}[c \in \Phi_j \mid c \in \mathcal{C}_k]$ ,

we have:

$$\begin{aligned} & \mathbb{E}(\mathbb{1}[\text{Detect}_k(x, \text{Att}(x, c)) = 0] \mid c \in \mathcal{C}_k) \\ &= 1 - \sum_{\Phi_j \in \mathcal{P}^{\Gamma_{\text{ERG}}}} (r_j \cdot \mathbb{E}[\text{Detect}_k(x, c_i) \mid c_i \in \Phi_j]) \\ &= 1 - \sum_{\Phi_j \in \mathcal{P}^{\Gamma_{\text{ERG}}}} (r_j \cdot \sum_{c_i \in \Phi_j} (\pi_i \cdot \mathbb{E}[\text{Detect}_k(x, c_i)])) \\ &= 1 - \sum_{\Phi_j \in \mathcal{P}^{\Gamma_{\text{ERG}}}} (r_j \cdot \sum_{c_i \in \Phi_j} (\pi_i \cdot \epsilon_{\text{pos}})) = 1 - \epsilon_{\text{pos}} \end{aligned}$$

□

In other words, when distribution consistency is satisfied, the distribution of the watermarking detectable N-gram feature among each equivalent space does not differ from the distribution among the whole code space  $\mathcal{C}$ . Therefore, after the obfuscation, the watermark detector cannot distinguish the watermarked code from benign code samples selected from the human-written space, making the "afterward FNR" increase to a very high level of  $1 - \epsilon_{\text{pos}}$ , effectively nullifying watermark detection.

The above impossibility theorem relies on the condition *large enough*  $t$ . To better understand the efficiency of our attack, with the mixing time  $t_i(\epsilon)$  defined in Appendix C.1, we further prove that:

**Theorem E.2 (Lower Bound of Attacked FNR).** With Assumption E.1 satisfied, given  $t \geq t_s(\epsilon)$ ,

in which  $c_s$  is the original watermarked code and also the starting point of the random walk, there exists an ergodicity rule set  $\Gamma_{\text{ERG}}$  that for all N-gram-based schemes with a false positive rate  $\epsilon_{\text{pos}}$ , the  $\text{RandomWalk}(c_s \mid c_s \in \mathcal{C}_k, \Gamma_{\text{ERG}}, t)$  algorithm can perform as an attacker and  $(1 - \epsilon - \epsilon_{\text{pos}})$ -break the watermarking.

*Proof of Theorem E.2.* Given  $t \geq t_s(\epsilon)$  achieved, in the worst case, the whole probability distribution is biased toward  $\text{Detect}_k(\cdot) = 1$ . Setting  $\Delta_i := |p(\cdot)_i - \pi_i|$  and  $D_i := \text{Detect}_k(x, c_i)$ , we have:

$$\begin{aligned} \sum_{c_i \in \Phi} (p(c_s, t)_i \cdot D_i) &= \sum_{\substack{c_i \in \Phi \wedge \\ D_i=1}} p(c_s, t)_i \\ &\leq \sum_{\substack{c_i \in \Phi \wedge \\ D_i=1}} [\pi_i + \Delta_i] = \sum_{\substack{c_i \in \Phi \wedge \\ D_i=1}} \Delta_i + \sum_{c_i \in \Phi} [\pi_i \cdot D_i]. \end{aligned}$$

In the worst case, the bias increases the probabilities of the  $D_i = 1$  cases, with a symmetric reduction on  $D_i = 0$  cases. Therefore:

$$\sum_{\substack{c_i \in \Phi \wedge \\ D_i=1}} \Delta_i \leq \frac{1}{2} \sum_{c_i \in \Phi} \Delta_i = \|p(\cdot) - \pi\|_{\text{TV}} \leq \epsilon.$$

Substituting into the previous calculation, with  $t \geq t_s(\epsilon)$ , we get:

$$\begin{aligned} &\mathbb{E}(\mathbb{1}[\text{Detect}_k(x, \text{Att}(x, c_s)) = 0] \mid c_s \in \mathcal{C}_k) \\ &= 1 - \sum_{\Phi_j \in \mathcal{P}^{\Gamma_{\text{ERG}}}} (r_j \cdot \mathbb{E}[\sum_{c_i \in \Phi_j} (p(c_s, t)_i \cdot D_i)]) \\ &\geq 1 - \sum_{\Phi_j \in \mathcal{P}^{\Gamma_{\text{ERG}}}} (r_j \cdot \mathbb{E}[\epsilon + \sum_{c_i \in \Phi_j} (\pi_i \cdot D_i)]) \\ &= 1 - \epsilon - \epsilon_{\text{pos}} \end{aligned}$$

□

Note that as "afterward FNR," the  $1 - \epsilon - \epsilon_{\text{pos}}$  is still too high to be an acceptable value, for the time complexity calculation shows in Theorem F.4 that  $t_s(\epsilon)$  has a linear dependence on  $\ln \epsilon^{-1}$ , meaning that  $\epsilon$  can be set to a small enough value with a slight affection on the  $t_s(\epsilon)$ .

## F Cost Analysis of Attacks

### F.1 Overall Analysis

In other sections, we prove that when  $t$  exceeds the mixing time and distribution consistency is satisfied, the robustness of the watermarking scheme can be effectively broken.

In this section, we estimate the order of this *sufficiently large*  $t$  and demonstrate that an attack tends to incur significantly lower time and space costs compared to the cost of code generation.

We emphasize that the time and space complexity should be various for different designs of the rule set. Therefore, we add intuitive and implementable additional definitions to simplify the calculation. The following analysis may not apply universally. Instead, our result should be considered an approximate solution applicable to general scenarios.

Our attack proceeds by first dividing the watermarked code into segments or code blocks. These segments only need to be sufficiently long to satisfy distribution consistency. Based on our experiments, this required length is relatively small. In fact, all three watermarking schemes lost the detection ability consistently during our experiment, and the 90-th percentile of segment lengths was 275.0, which can serve as a reference value for setting  $l$ . Next, for each sub-segment, we perform iterative random walks until the mixing time is reached.

For the following proof, we set several symbols:

- Given  $\Phi \in \mathcal{P}^{\Gamma_{\text{ERG}}}$ , let the space size  $N = |\Phi|$ .
- Let the length of the watermarked code be  $L$  and the length of the split code segment be  $l$ .
- Let the receptor number of the code segment be  $\wp$ . As for the concept *receptor*, it represents the code component that is targeted by rules. E.g., code variables for variable modification rule, comments for comment modification rule, etc.
- Let the out-degree of each vertex in the code transformation graph be  $d$ .

We can follow the general assessment that the complexity of the action derivation algorithm R is  $O(l)$  and Transform is  $O(1)$ , for both time complexity and space complexity, since R needs to iterate the code and find the applicable actions, and each action performed by Transform only leads to modifications of constant order. Moreover, since the new modifications after each call of Transform are of constant order and the R in the next round only needs to analyze the updates, we can assess the complexity of R as  $O(1)$  if not the initial call of R.

Therefore,  $O(l)$  is the overall space complexity, and  $O(\frac{L}{l} \cdot (l + t))$  is the time complexity. The key point is to assess the mixing time  $t$ .

In the following sections (Theorem F.4), we prove that  $t_i(\epsilon) \leq O(l^2 \ln \epsilon^{-1} + l^3)$ . In the case that  $L$  is relatively short and we do not split the code into segments ( $L = l$ ), the overall time complexity of our attack will be  $O(L^2 \ln \epsilon^{-1} + L^3)$ . When we consider the situation that  $L \gg l$  and we apply the split, the overall complexity will be  $O(L \cdot (l \cdot \ln \epsilon^{-1} + l^2)) = O(L)$ , since  $L \gg l$  (as shown in our experiments,  $l$  does not need to be large).

Compared with code generation, nowadays, transformer-based generative models can be considered as having time and space complexity  $O(L^2)$  with a well-known high constant factor. Since our attack generally only needs lightweight CPU computation and memory storage, our algorithm with  $O(l)$  space and  $O(L)$  time needed requires much less computation than code generation. Even if we do not split the watermarked code during the attack and obtain  $O(L^2 \ln \epsilon^{-1} + L^3)$  time complexity, the algorithm still tends to be faster than LLM generation when we have small scale  $L$  due to generative model's high constant factor.

## F.2 Congestion and Canonical Paths

To assess the mixing time, we introduce classic and effective tools from Markov chain theory, *congestion* and *canonical paths* (Jerrum, 2003). Given code transformation graph  $G_c^{\Gamma_{\text{ERG}}} = (V, E, W)$ , vertex indices  $i, j, I, J \in \mathbb{N}$ , we have:

- **Canonical Path:** for any pair  $v_i, v_j \in V$ , define a canonical path  $\zeta_{ij} = (v_i = \dots = v_j)$  from  $v_i$  to  $v_j$  through directed edges, and let  $\Upsilon := \{\zeta_{ij} \mid v_i, v_j \in V\}$  be the set of all canonical paths.
- **Congestion:** The *congestion*  $\varrho$  of the graph is defined by:

$$\varrho(\Upsilon) = \max_{e_{IJ} \in E} \left\{ \frac{\sum_{i,j: \zeta_{ij} \text{ uses } e_{IJ}} \pi_i \pi_j |\zeta_{ij}|}{\pi_I \bar{P}_{IJ}} \right\}$$

- **Congestion and Mixing Time:** Denoting  $\varrho := \varrho(\Upsilon)$ , the mixing time  $t_i(\epsilon)$  is bounded by:

$$t_i(\epsilon) \leq 2\varrho(2 \ln \epsilon^{-1} + \ln \pi_i^{-1}).$$

Canonical paths and congestion provide us with an elegant approach to bound mixing times, in which we are allowed to flexibly define the form of our canonical paths.

## F.3 Graph Definitions

In this section, we begin the mixing-time calculation. We define the concept *receptor*, which means the element that receives the code transformation, for example, variables for the rule "Randomly rename variables" and dead code block for "Delete a dead code snippet" in Table 2.

Given the notion that the size of  $\Phi \in \mathcal{P}^{\Gamma_{\text{ERG}}}$  is  $N$  and the out-degree of transformation graph is  $d$ , we stress the following additional definitions, which are intuitive and implementable, to simplify the calculation.

We clarify that the following mixing time estimation is derived under these specific assumptions. While it is exact in such constrained settings, it can also serve as an approximate solution in more general scenarios.

**Code Length** When random walk on code segments inside the same transformation graph, all lengths of code are of the order  $\Theta(l)$ , i.e., the ratios among them are of the constant order. It is less reasonable if we obtain a result after a random walk that is too short or too long compared with the original watermarked code, and we also can constrain the range of code length during the random walk by the implementation of our rule set.

**Independent Receptors** All receptors in a code segment are satisfied: **1)** Non-overlapped. One element of code can only receive one rule. For example, in the rule set of Table 2, the variable names inside a dead code snippet can be modified by either the variable renaming rule or the dead code deletion rule, but not both. **2)** Pre-defined. At the initial stage, the rule algorithms will identify receptors with fixed positions. For example, the comment-adding rule will identify positions that can accept a new comment as its receptors, together with existing comments positions as candidates. At the following stages, it will only select a position without comments from these pre-defined positions for adding. **3)** State-changeable. Each receptor can be modified to have multiple states but will not disappear during the random walk. For example, after a transformation deletes a comment, the original position turns into a receptor of the comment addition rule.

Therefore, the receptor number  $\wp = \Theta(\beta \cdot l) = \Theta(l)$ , in which  $\beta \leq 1$  is a constant and denotes the expected number of receptors distributed per code token.

**Out-degree** The ratio between the out-degree values of two vertices in the same transformation graph is of the constant order. Because for different code segments with similar lengths, with the same rule set, the number of possible transformation actions should be of the same order. Therefore, we can denote out-degree  $d = \Theta(\alpha \cdot \wp) = \Theta(l)$ , in which  $\alpha \geq 1$  is a constant and denotes the expectation number of actions that rule set  $\Gamma_{\text{ERG}}$  can perform on each receptor location.

#### F.4 Mixing Time Calculation

In this section, we will discuss subexpressions separately and conduct the mixing time from congestion.

**Definition F.1 (Canonical Path for Code Transformation Graph).** For any pair  $v_i, v_j \in V$ , a canonical path from  $v_i$  to  $v_j$  satisfied:

- Shortest Path: The  $\zeta_{ij}$  can not be shorter than other paths from  $v_i$  to  $v_j$ , formally:

$$|\zeta_{ij}| = \min \{|\zeta| \mid \zeta = (v_i = \dots = v_j)\}.$$

Therefore, we have  $|\zeta_{ij}| = \Theta(\wp) = \Theta(l)$  since each receptor can be modified into the final stage within one shot.

- In Sequence Modification: For edges in each  $\zeta_{ij}$ , the corresponding receptors follow some queuing convention. The receptor's rank in the queue is monotonically increasing along the canonical path.

Note that the *in sequence modification* definition is implementable. For example, the receptors are sorted by their hunk positions from front to back, and then we can execute the modifications for sequenced receptors one by one.

**Theorem F.1 (Transition Matrix Elements are of  $\Theta(\frac{1}{d})$ ).** For all  $e_{ij} \in E$ , the corresponding  $\vec{P}_{ij} = \Theta(\frac{1}{d})$ .

*Proof of Theorem F.1.* From Definition C.2, which defines an uniform and random algorithm to pick the next vertex during random walk, given out-degrees are of the order  $\Theta(d)$ , it follows immediately that for all  $e_{ij} \in E$ ,  $\vec{P}_{ij} = \Theta(\frac{1}{d})$ .  $\square$

**Theorem F.2 (Stationary Distribution Elements are of  $\Theta(\frac{1}{N})$ ).** For stationary distribution  $\pi$ , it satisfied:

$$\forall i \in \{1, 2, \dots, N\}, \pi_i = \Theta\left(\frac{1}{N}\right)$$

*Proof of Theorem F.2.* First, from *inverse rule* property defined in Definition 3.2, for each vertex in the  $G_c^{\Gamma_{\text{ERG}}}$ , its out-degree is equal to in-degree. Then, we assume after  $t$  steps of random walk:

$$\forall v_i \in V, p(c, t)_i = \frac{d_i}{\sum_{v_k \in V} d_k},$$

in which  $d_i$  denotes the out-degree of  $v_i$ . The denominator in this equation means the sum of all vertices' out-degrees. Considering further step  $t + 1$ , for all  $v_i \in V$ , we have:

$$\begin{aligned} p(c, t+1)_i &= \sum_{e_{ji} \in E} p(c, t)_j w_{ji} \\ &= \sum_{e_{ji} \in E} \frac{d_j}{\sum_{v_k \in V} d_k} \cdot \frac{1}{d_j} = d_i \cdot \frac{1}{\sum_{v_k \in V} d_k} \\ &= p(c, t)_i. \end{aligned}$$

We can notice that after the assumption is satisfied, the random walk falls into its stationary. Due to the uniqueness of the stationary distribution, we can confirm that for all  $v_i \in V$ :

$$\pi_i = \frac{d_i}{\sum_{v_k \in V} d_k} = \Theta\left(\frac{d}{|V| \cdot d}\right) = \Theta\left(\frac{1}{N}\right).$$

$\square$

**Theorem F.3 (Canonical Path Number Uses Particular Edge are of  $O(N)$ ).** Given the  $\zeta$  definition in Definition F.1, for each  $e_{IJ} \in E$ , we have:

$$|\{\zeta_{ij} \mid \zeta_{ij} \text{ uses } e_{IJ}\}| = O(N)$$

*Proof of Theorem F.3.* Given definition in Definition F.1, we further define the degrees of freedom on receptors ranked 1 to  $\wp$  are  $d_1^{\text{free}}, \dots, d_{\wp}^{\text{free}}$ . Therefore, the space size  $N = \prod_{k=1}^{\wp} d_k^{\text{free}}$ .

For each path uses  $e_{IJ}$ , if the corresponding receptor of  $e_{IJ}$  is ranked  $K$ . We can notify that the sub-path before  $e_{IJ}$  is limited to only involve the modifications on receptors ranked 1 to  $K$ , and vice versa, the sub-path after  $e_{IJ}$  can only involve receptors  $K$ -th to  $\wp$ -th.

Therefore, the degree of free of  $v_i$ , i.e., the path's starting vertex, is not bigger than  $\prod_{k=1}^K d_k^{\text{free}}$ , and for  $v_j$  is not bigger than  $\prod_{k=K}^{\wp} d_k^{\text{free}}$ . We consider all combinations of starting vertices and ending vertices to get the following:

$$\begin{aligned} &|\{\zeta_{ij} \mid \zeta_{ij} \text{ uses } e_{IJ}\}| \\ &\leq \prod_{k=1}^K d_k^{\text{free}} \cdot \prod_{k=K}^{\wp} d_k^{\text{free}} = O(N). \end{aligned}$$

$\square$

**Theorem F.4 (Mixing Time Calculation).** Given initial code segment  $c$ , segment length is of  $\Theta(l)$ , the mixing time  $t_i(\epsilon)$  of random walk satisfied:  $t_i(\epsilon) \leq O(l^2 \ln \epsilon^{-1} + l^3)$ .

*Proof of Theorem F.4.* Given Theorem F.3, we have:

$$\ln N \leq \ln \left[ \max \left\{ d_1^{\text{free}}, \dots, d_\phi^{\text{free}} \right\}^\phi \right] = O(l).$$

Together with Definition F.1, Theorem F.1, and Theorem F.2, we have:

$$\begin{aligned} t_i(\epsilon) &\leq 2\varrho(2 \ln \epsilon^{-1} + \ln \pi_i^{-1}) \\ &= O \left( \frac{N \cdot \frac{1}{N} \cdot \frac{1}{N} \cdot \partial \varphi}{\frac{1}{N} \cdot \frac{1}{d}} \cdot (2 \ln \epsilon^{-1} + \ln N) \right) \\ &= O(l^2 \cdot (2 \ln \epsilon^{-1} + \ln N)) \\ &= O(l^2 \ln \epsilon^{-1} + l^3), \end{aligned}$$

which means following our code transformation model, a rapid mixing (Guruswami, 2000) would be achieved. Together with our analysis in Appendix F.1, this time complexity leads to a significantly lower runtime cost than LLM-based code generation.  $\square$

## G Impossibility of Impossibility

In this section, we demonstrate the unfeasibility of the previous work’s impossibility proof (Zhang et al., 2024) on program generation tasks, in which the authors made two critical assumptions:

- **Quality oracle:** The attacker has access to a *quality oracle*, allowing efficient verification of whether a disturbed result still meets the original prompt’s requirements, i.e., belonging to the high-quality space.
- **Perturbation oracle:** The attacker has a *perturbation oracle*, which can efficiently generate random disturbances to responses while maintaining their quality. The perturbation oracle relies on a quality oracle to validate whether a randomly perturbed answer still belongs to the high-quality space.

Under these assumptions, the authors proved that no watermarking scheme could retain its robustness. They argued that these assumptions are practically reasonable because of the heuristic notion that "verification is easier than generation." However, we argue that in programming generation tasks, these assumptions, especially the existence of a quality oracle, do not hold either theoretically or practically.

**Theoretical Impossibility** Specifically, unlike natural language scenarios where verification might indeed be simpler than generation, verifying correctness in programming generation needs to observe the running behavior and is fundamentally challenging. The quality oracle described would need to check whether the generated code precisely matches a prompt describing the required program behavior. However, this universal verification task is theoretically impossible due to fundamental limitations rooted in computability theory. It is deeply connected to the well-known *halting problem* (Kleene, 1952). The halting problem demonstrates that no algorithm can universally determine whether an arbitrary program halts (terminates) or continues indefinitely for all possible inputs.

Building upon the halting problem, Rice’s theorem (Rice, 1953) generalizes this concept further, asserting that virtually all meaningful (non-trivial) behavioral properties of programs, such as correctness, termination, or compliance with certain specifications, are undecidable. In simpler terms, Rice’s theorem implies that there is no universal verifier capable of reliably assessing program behavior across all possible cases. This limitation is fundamental, not just an occasional exception.

Therefore, even for a seemingly straightforward prompt such as "Generate a program that halts," constructing an accurate quality oracle would involve solving the halting problem itself, which is known to be impossible. Extending this reasoning, we conclude that most practical programming prompts inherently involve verifying non-trivial semantic properties, making a universal quality oracle fundamentally unattainable.

For a more practical example, consider a prompt such as "generate a Python server demo." Implementing a quality oracle for this task would require determining whether a perturbed program is a valid Python server and verifying if its runtime performance (e.g., latency, throughput, correctness under various conditions) remains consistent with the original unperturbed program. This form of comprehensive runtime and semantic verification is also theoretically impossible, as it would necessitate solving undecidable problems.

**Practical Impossibility** Even if we add many restrictions (e.g., timeout, using LLMs to generate test cases, considering only codes that are runnable in Att’s sandbox, etc.) to make it theoretically im-

plementable, achieving the quality oracle is still an open question and tends to consume large amounts of computational resources, leading to an attack that is too inefficient to be realistic.

Many works highlighted the difficulty of automating generating test cases from natural language business rules. Even though state-of-the-art generative models can be leveraged, there are still overwhelming challenges, including high intellectual demand, uncontrollable and intractable outputs, limited domain knowledge in pre-training models, test-oracle problems, rigorous evaluations, and high implementation costs in the real-world applications (Corriveau et al., 2014; Xue et al., 2024; Wang et al., 2024).

We can also estimate the difficulty from past practice. An acceptable code evaluation is both hard and resource-consuming, even with ground truth code given. EvalPlus (Liu et al., 2023) needs a state-of-the-art model to generate around 30 seed inputs and generate 1000 additional inputs using a one-hour budget, for each task. LiveCodeBench (Jain et al., 2025) needs to generate other code segments as input generators, and there are 2 random and 4 adversarial input generators needed for each task. Considering the quality oracle, with only a natural-language description and reference code as input, it can only be harder without ground truth code to guide the test generation. CodeT (Chen et al., 2023) needs to generate around 50 other candidate solutions (1000 for harder tasks) to perform cross-validation, and they generate 100 test cases for each task. All works above are required to call state-of-the-art generative models.

Therefore, this is the opposite of the previous work’s assumption that, even if we add restrictions to get around theoretical impossibilities, the code evaluation would be likely to consume similar or more computational resources than a generation, leading to a theoretically impossible and low cost-effectiveness attack. This is also the motivation of our *code transformation rule* modeling.

In summary, to build a *quality oracle*, there needs to be a theoretically impossible evaluator that tests whether codes fulfill requirements from the prompt. Even considering multiple restrictions are added to bypass the theory barrier, it still tends to require higher computational resources to evaluate the code than code generation. We do not deny the feasibility of applying more advanced code perturbation methods and semantics checkers (e.g., use LLMs to re-generate code and use EvoSuite

(Fraser and Arcuri, 2011) to ensure equivalence where possible). However, any such attempt can only be seen as performing a random walk on a larger rule-based equivalent space but not the entire high-quality space, falling into the modeling in this paper, and an analysis of distribution consistency is needed.

## H Discussion: Semantic Awareness of Code Watermarking

In this section, we will discuss a possible path to improving the robustness of code watermarking schemes, i.e., semantic awareness.

In Subsection 3.3, we analyze that if the randomized watermarking algorithms operate at the N-gram level without the awareness of syntax or semantics, the distribution consistency is more achievable since the transformation rules are generally syntax- or semantics-based.

On the other hand, in Appendix G, we also show that it is impossible to random walk or obtain semantic equivalence among the entire high-quality space  $\Phi^F$ , since they all rely on semantics comparison or semantics understanding, which would run into the undecidability barrier captured by Rice’s theorem (Rice, 1953), and leaves space for watermarking robustness.

Suppose we can introduce bias in the semantics of the generated code and detect the semantic features. In that case, attackers will need similar or higher computational costs to disturb the watermark than code generation.

As an example, one semantics-related component would be third-party API calling. Containing rich semantics, disruption on it needs an extensive understanding of semantics (e.g., transforming `np.square(matrix)` to `matrix**2` needs the knowledge of NumPy API). With obfuscators nowadays, third-party API calls generally remain even after obfuscation due to their semantics-related nature.

Consider a task that generates long code with multiple third-party API calls. If we have a watermarking scheme that can 1) apply hidden strategies on how to select and call third-party APIs and 2) detect these hidden patterns in the sequence of API calls, the robustness under attack would be kept to a considerable extent. The difficulty of attacking this watermarking scheme would be:

- For attackers with a rule set that cannot disturb the API call, distribution consistency (As-

sumption 3.1) would not hold since programs from each rule-based equivalent space share the same API calling feature, and the whole space can be considered watermarked.

- For attackers that are dedicated to disrupting the API-call features, a similar level of third-party API understanding compared with the code generation model is needed. Either dynamically disrupting API calls with a generative model or mining a comprehensive API transformation rule set would be costly and difficult to perform precisely. Meanwhile, a post-evaluation scheme (like the quality oracle in previous work (Zhang et al., 2024)) to bound the quality of transformed code would also be costly, as discussed in Appendix G.

Therefore, we consider a code watermarking scheme with high semantics awareness as robust.

### H.1 A Proof-of-Concept Design

To make the path above concrete, we sketch a proof-of-concept scheme that operates over the sequence of third-party API calls in the generated program rather than over token N-grams. Algorithm 3 gives the pseudocode.

During generation, when the model is about to emit an API call, the scheme first identifies semantically valid candidate APIs for the current context at runtime. The most recent  $N - 1$  API calls are then hashed to produce a seed that partitions these candidates into green and red sets, analogous to the token-level green list, and the model is biased toward a green-list choice; the first  $N - 1$  calls of a program are sampled without bias, following the same convention as the token-level scheme (Appendix A). For example, when both satisfy the required behavior, the model may prefer `pandas.isna(x)` over its documented alias `pandas.isnull(x)`. During detection, the same keyed hash scores the observed API choices from the  $N$ -th call onward, and a z-score is computed from the number of scored calls assigned to the green set.

This design survives the obfuscation attacks considered in this paper. Obfuscators are generally syntax-based and do not perform semantic substitution between third-party APIs, so they preserve the selected API choices and their semantic call order. Under such an obfuscator, every program in a rule-based equivalent space shares the same API-call feature, so the space is watermark-homogeneous

with respect to the detector: distribution consistency (Assumption 3.1) fails, and the detector retains discriminative power. Substituting one third-party API for a semantically equivalent alternative would instead require API-level knowledge that existing rule sets lack, with the costs discussed above.

Note that showing the possibility of code watermarking robustness is not within the scope of this paper, and high implementation costs might accompany the path we discussed.

## I Original Performances of Watermarking Schemes

Before studying robustness, we first analyze the original performance of the three selected watermarking approaches without an attack.

For distortionary watermarking schemes, i.e., SWEET and WLLM, the results on LLaMA against Python-Minifier and JS Obfuscator are shown in Figure 5. The DeepSeek results are shown in Figure 8. The left-side sub-figures show that the original watermarking can achieve good detection performance. For LLaMA, all four code benchmarks have parameter combinations that lead to AUROC scores higher than 0.9 while maintaining more than 80 % of the same model’s original non-watermarked performance. For example, under the WLLM setting ( $\delta = 4.00, \gamma = 0.10$ ), AUROC = 0.97 while maintaining 84 % non-watermarked performance. For DeepSeek-Coder, all watermarking schemes also achieve AUROC  $\simeq 0.8$  while maintaining 80 % non-watermarked performance.

The results for SynthID are shown in Figure 6, as well as Figure 9 in the appendix. SynthID causes no obvious code-quality degradation for either model, due to its non-distortionary nature (Dathathri et al., 2024). For all experiments on SynthID, the average Pass@1 decrease is  $-1.37 \times 10^{-3}$ . Considering detection performance under different settings, AUROC increases with temperature. The same AUROC trend can be observed from both models. With the temperature increasing from 0.25 to 1.25, among four benchmarks and two models, the average AUROC score increases from 0.56 to 0.91.

### Low-entropy Issue in Code Watermarking:

From this result, we reaffirmed the low-entropy issue in code watermarking, which was also discussed in previous work (Lee et al., 2024). To improve code quality, when generating code, the generation parameters are usually set to reduce the randomness, e.g., lower temperature or lower TopK.

---

**Algorithm 3:** Proof-of-concept semantics-aware watermark over third-party API calls

---

**Input:** Model  $M$ , prompt  $x$ , key  $k$ , green fraction  $\gamma$ , logit bias  $\delta$ , context length  $N$ , detection threshold  $\tau$

```
1 Embedding:
2 Init generated code  $c \leftarrow ()$ , API-call history  $A \leftarrow ()$ 
3 while generation is not finished do
4   if  $M$  is about to emit a third-party API call then
5     if  $|A| \geq N - 1$  then
6       Init candidate set  $C \leftarrow$  semantically valid APIs for the current call site
7       Init seed  $s \leftarrow H(\text{last } N - 1 \text{ elements of } A, k)$ 
8       Partition  $C$  by  $s$  into a green set  $G_s$  with fraction  $\gamma$  and a red set  $C \setminus G_s$ 
9       Add bias  $\delta$  to the logits of APIs in  $G_s$ ; sample the call  $a$ 
10    else
11      Sample the call  $a$  without bias
12    Append the tokens of  $a$  to  $c$ ; append  $a$  to  $A$ 
13  else
14    Sample the next token from  $M$  and append it to  $c$ 
15 Return  $c$ 
16 Detection (given code  $c'$  and key  $k$ ):
17 Init  $A' \leftarrow$  the sequence of third-party API calls extracted from  $c'$ , green count  $g \leftarrow 0$ , scored count
    $m \leftarrow |A'| - N + 1$ 
18 for  $i \leftarrow N$  to  $|A'|$  do
19   Init candidate set  $C \leftarrow$  semantically valid APIs at the call site of  $A'_i$ 
20   Init seed  $s \leftarrow H(A'_{i-N+1} \dots A'_{i-1}, k)$ 
21   Partition  $C$  by  $s$  into a green set  $G_s$  with fraction  $\gamma$  and a red set  $C \setminus G_s$ 
22   if  $A'_i \in G_s$  then
23     Set  $g \leftarrow g + 1$ 
24 Return watermarked iff  $z = (g - m\gamma) / \sqrt{m\gamma(1 - \gamma)} > \tau$ 
```

---

DeepSeek even directly applies greedy search when they evaluate on HumanEval-X (Guo et al., 2024). This exacerbates low-entropy characteristics in code generation and makes watermark detection harder, for the mainstream watermarking method requires higher entropy for better detection (Lee et al., 2024; Dathathri et al., 2024). In Figure 6, we can observe this trend: as temperature decreases, code quality improves, but detection AUROC drops sharply. This provides additional evidence that N-gram-based watermarking is unsuitable for code content.

## J Ideal Watermarking Scheme Under Attack

We construct an ideal watermarking scheme with unrealistically high detection capability and quality preservation, trying to answer a question: If we keep developing N-gram-based watermarking and

|                      | Pass@1 | AUROC  |
|----------------------|--------|--------|
| Ideal Watermarking   | 0.9634 | 0.9747 |
| Attacked w/ UglifyJS | -      | 0.5085 |

Table 3: Ideal watermarking scheme performances before/after attack. The Pass@1 score after the attack is omitted because it is ensured to be identical with before.

its detection ability increases dramatically, would high robustness eventually emerge?

Algorithm 4 shows the generation of our watermarking scheme. The `isMarked` is a hash algorithm that will return 1 when a *marked* 5-gram is provided as input. We randomly selected half of the 5-grams as marked. For each task, it repeatedly generates  $t_{\text{gen}}$  times (See line 3;  $t_{\text{gen}} = 500$  in our experiment) with randomized generative model  $M$ , which is a regular model without any watermarking related components. For each generation, we use the test suite provided by the benchmark to filter

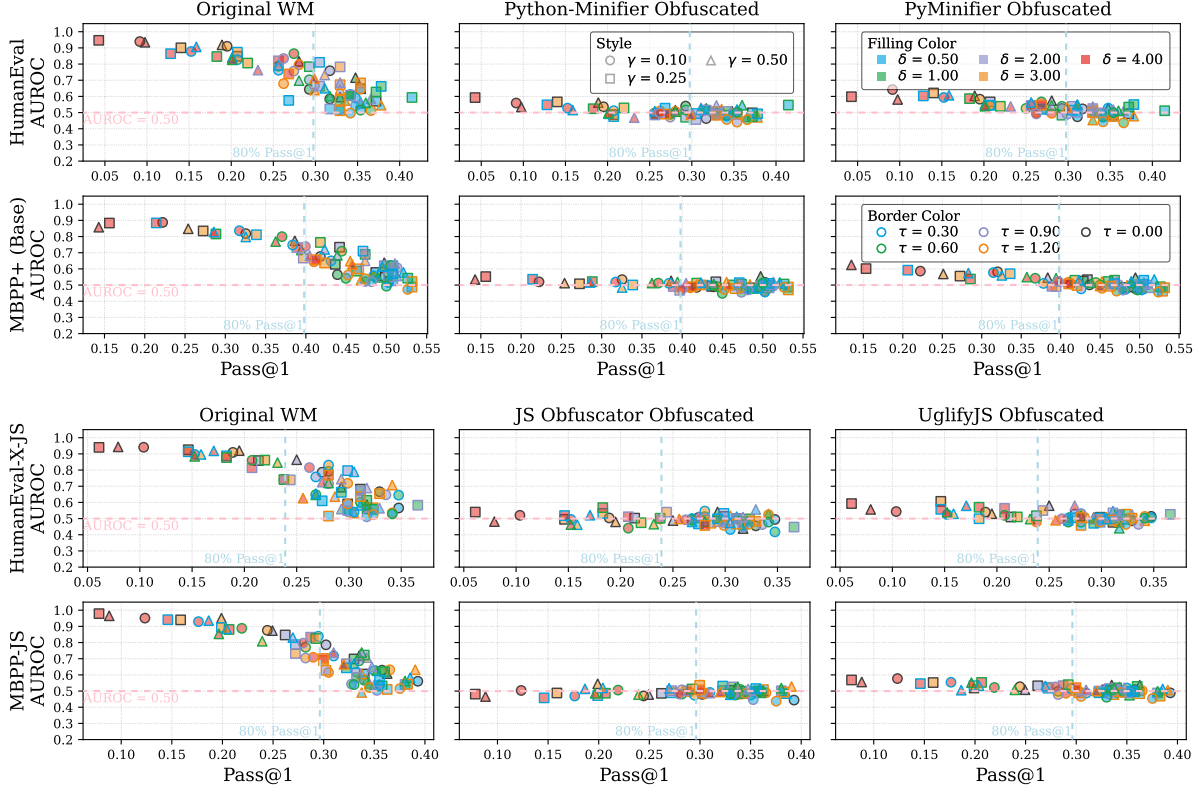


Figure 8: WLLM and SWEET (Kirchenbauer et al., 2023; Lee et al., 2024) on DeepSeek-Coder-33B-Base (Guo et al., 2024), before and after attack. The upper panel shows the effect of our attack on Python, and the lower panel shows the effect of our attack on JavaScript. The y-axis is AUROC and the x-axis is Pass@1; colors and markers denote hyperparameters. For LLaMA results, see Figure 5.

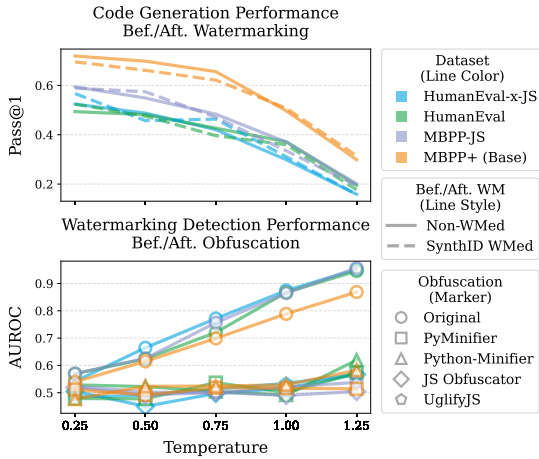


Figure 9: Upper: DeepSeek Coder Pass@1 changes with temperature. Comparison between SynthID (Dathathri et al., 2024) watermarked and non-watermarked code generation. Lower: SynthID detection AUROC changes with temperature. Comparison between original SynthID watermarked code and obfuscated code. For results from the LLaMA 3.1, see Figure 6.

out code that does not satisfy the prompt requirements (line 5). Finally, if any of the generations can pass the test suite, we select the code with the highest ratio of marked 5-gram as the returned code (line 14).

For detection, we use the same `isMarked()` algorithm to calculate the ratio of marked 5-gram for each code under detection, then calculate corresponding  $z$ -score as the detection metric.

We claim that this watermarking algorithm has two properties that lead to unrealistically high detection capability and quality preservation, which realistic N-gram-based watermarking schemes are unlikely to achieve even in the future. We aim to exploit the representational power of N-gram features as much as possible to combat our attacks.

- **Global Awareness:** Compared with realistic N-gram-based watermarking schemes that generally can only perform greedy algorithm per token, this scheme directly selects response with highest final global metric.
- **Test Oracle Assistance:** As we analyzed in

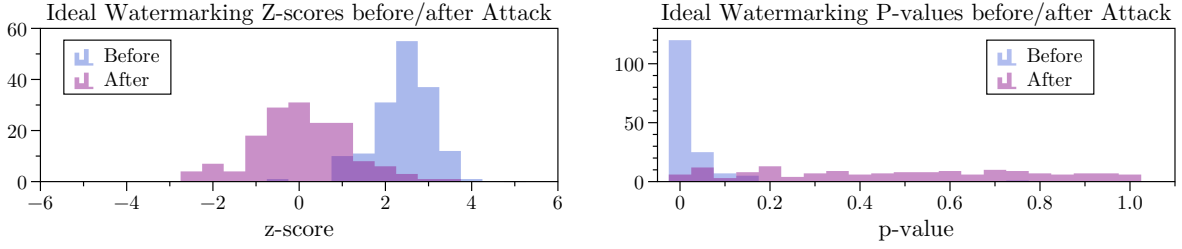


Figure 10: Left: Frequency distribution chart for z-scores of generated code from ideal watermarking scheme, before/after UglifyJS-based attack. Right: Frequency distribution chart for p-values.

---

**Algorithm 4:** Ideal Watermarking

---

```

//  $\mathcal{V}$  is the space of tokens (vocabulary)
//  $\text{isMarked} : \mathcal{V}^5 \rightarrow \{0, 1\}$  is the algorithm
// that can distinguish whether a 5-gram
// is marked, returning 1 denotes a marked
// 5-gram

Input: Generative Model:  $M : \mathcal{X} \rightarrow \mathcal{C}$ ,
        Prompt:  $x \in \mathcal{X}$ ,
        Generation Times:  $t_{\text{gen}} \in \mathbb{N}$ ,
        Benchmark Given Test Suite:  $T_{\text{ben}}$ 

Output: Watermarked Code:  $c_{\text{wm}} \in \mathcal{C}$ 
1 Init Highest Watermarked Ratio  $r_{\text{final}} \leftarrow 0$ 
2 Init Watermarked Code  $c_{\text{wm}} \in \mathcal{C}$ 
3 for each  $i$  in  $1, 2, \dots, t_{\text{gen}}$  do
4   Set  $c_i \leftarrow M(x)$ 
5   if  $T_{\text{ben}}(x, c_i) = 1$  then
6     Init Watermarked Ratio  $r_{\text{wm}} \in \mathbb{R}$ 
7     Set  $r_{\text{wm}}$  to the ratio of  $c_i$ 's 5-grams
       which let  $\text{isMarked}()$  return 1
8     if  $r_{\text{final}} < r_{\text{wm}}$  then
9       Set  $r_{\text{final}} \leftarrow r_{\text{wm}}$ 
10      Set  $c_{\text{wm}} \leftarrow c_i$ 
11 if  $r_{\text{final}} = 0$  then
12   Return No Result
13 else
14   Return  $c_{\text{wm}}$ 

```

---

Appendix G, high costs are needed for implementation of test suite (theoretically impossible for a generalized one). However, in this ideal scheme, we directly leverage the proprietary test suite in the benchmark to filter out unpassed generation, leading to an unrealistically high quality remaining.

Following Subsection 4.1 settings, we use HumanEval-X-JS (Zheng et al., 2023) and leverage UglifyJS-based attack on this ideal watermarking scheme. The result is in Table 3, and we gather

the z-scores and p-values of the tasks, drawing the distributions in Figure 10. We remind that our null hypothesis is z-scores following  $\mathcal{N}(0, 1)$ , meaning that we cannot discriminate the samples under detection is whether directly sampled from the distribution of the whole code space  $\mathcal{C}$  or not.

We can notice that this ideal scheme has nearly perfect detection and quality preservation, i.e., 0.9747 AUROC score and 0.9634 Pass@1 (158/164 passed). The left subfigure also shows of Figure 10 that the z-score distribution of the ideal watermarked samples is significantly biased from the standard normal distribution, whose significance can be confirmed by p-values (right subfigure of Figure 10), in which 72.78% samples have p-values lower than 0.02 and 87.97% lower than 0.05.

Although the scheme performed ideally, the excellent detection ability is not accompanied by better robustness. After the attack, with a Pass@1 score proven to be maintained at 96.34%, the 0.5085 AUROC score shows that the detection effect after the attack is like flipping a coin. Meanwhile, we can notice from Figure 10 that the z-scores distributed around zero, and only 7.59% (87.97% before) cases have p-value lower than 0.05 and can be seen as significant.

From the results, we can conclude that even if we exploit the representational power of N-gram features far deeper in the future, the N-gram-based watermarking still has weak robustness confronting our attack.

## K Distribution Consistency Test on the Idealized Watermarking Scheme

We directly test the core assumption behind our theory: distribution consistency (Assumption 3.1). Starting from 158 highly watermarked code segments produced by the idealized scheme in Appendix J, we construct approximate rule-based

**Algorithm 5:** Equivalent Space

**Input:** Seed Code:  $c \in \mathcal{C}$ ,  
 Normalizer:  $\text{norm} : \mathcal{C} \rightarrow \mathcal{C}$ ,  
 De-normalizer:  $\text{deNorm} : \mathcal{C} \rightarrow \mathcal{C}$ ,  
 Expected Space Size:  $n \in \mathbb{N}$

**Output:** Equivalent Space:  $\Phi \subseteq \mathcal{C}$

```

1 Init Equivalent Space  $\Phi \leftarrow \emptyset$ 
2 Init Normalized Code  $c_{\text{norm}} \leftarrow \text{norm}(c)$ 
3 while  $|\Phi| < n$  do
4   Init  $c' \leftarrow \text{deNorm}(c_{\text{norm}})$ 
5   if  $c_{\text{norm}} = \text{norm}(c')$  then
6     Set  $\Phi \leftarrow \Phi \cup \{c'\}$ 
7 Return  $\Phi$ 

```

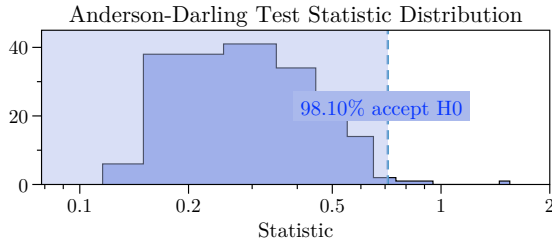


Figure 11: Anderson-Darling statistics distribution for testing whether each approximate equivalent space follows  $\mathcal{N}(0, 1)$ ; the blue line marks significance level 5.0.

equivalent spaces and test whether their internal  $z$ -score distributions match the standard normal distribution of the broader code space.

Algorithm 5 shows the construction of each equivalent space. Specifically, we use UglifyJS as a code normalizer (we did not find an effective code normalizer) and gpt-4o (OpenAI et al., 2024) as a randomized de-normalizer. The algorithm repeatedly de-normalizes the normalized seed code, trying to generate code samples that belong to the same equivalent space as the seed code. If the de-normalized code has the same normalized form as the seed code (line 5), it must belong to the same equivalent space as the seed code, i.e., the normalizer has the ability to transform one into another. The algorithm will repeat until the size of constructed equivalent space reaches the expected size (line 3) (30 in practice). In this way, we use a randomized de-normalizer to select samples from equivalent spaces.

Using the Anderson–Darling test, Figure 11 shows that 98.10% of equivalent spaces fail to reject the null hypothesis at the conventional 5.0% significance level. The remaining three outlier spaces do not indicate signals that the watermark

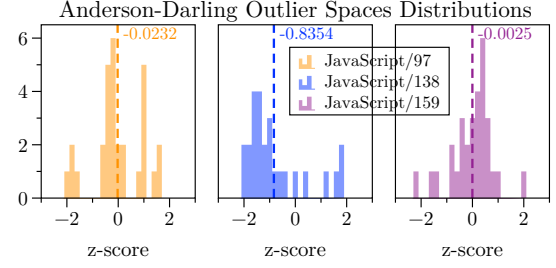


Figure 12: Z-score distributions of the few outlier spaces, with dashed lines marking their means.

preserved either: one is biased toward lower  $z$ -scores, and the other two have means close to zero (see Figure 12). Even at the more permissive 15.0% significance level (Stephens, 1974), 83.54% still match  $\mathcal{N}(0, 1)$ .

In addition, because we use UglifyJS as a normalizer even though obfuscators are not designed for code normalization, our results provide a conservative estimate of distribution consistency compared with using a dedicated normalizer built from various implementable equivalent transformation rules. These results support that a practical rule set can satisfy distribution consistency to a high degree, leaving little room for the watermark signal to survive within the equivalent space.

## L A Theory-Grounded Random-Walk Obfuscator

Our theoretical attack (Theorem 3.3) is a randomized walk driven by an ergodic rule set, whereas the off-the-shelf tools in Subsection 4.1 apply their transformation rules along fixed paths. To connect the two directly, we implement a Python obfuscator that follows the random-walk construction of Subsection 3.2.

**Construction.** The obfuscator operates with reversible transformation rules, a no-op transition, and a bounded state space, matching the requirements of Definition 3.2. For example, variable renaming is made reversible by mapping each identifier’s initial name to a finite candidate set and allowing the inverse mapping back to the initial name. The no-op transition guarantees aperiodicity, and each walk is restricted to a fixed 2,000-character length interval around its seed program, which keeps the reachable state space finite. The transformation rules are largely adapted from Python-Minifier, extended with two additional sets of reversible rules that implement control-flow flattening and opaque-predicate insertion (i.e., advanced

rules). We run 100 transformation steps per program; every program that passed its benchmark tests before transformation continued to pass them afterward.

**Evaluation.** We apply the obfuscator to all effective settings from our experiments, defined as those retaining more than 80% of the non-watermarked performance while achieving an AUROC above 0.8. This yields 43 selected Python experimental settings—13 for WLLM, 24 for SWEET, and 6 for SynthID—with each watermarking configuration counted separately on each dataset, covering 11,118 watermarked code programs in total. To isolate the contribution of the advanced rules, we also run a variant of the obfuscator with control-flow flattening and opaque predicates disabled, and we report Python-Minifier on the same settings for comparison.

| Obfuscator                               | WLLM  | SWEET | SynthID | Overall |
|------------------------------------------|-------|-------|---------|---------|
| <i>Settings with AUROC in (0.4, 0.6)</i> |       |       |         |         |
| Random walk                              | 13/13 | 24/24 | 6/6     | 43/43   |
| w/o advanced                             | 13/13 | 23/24 | 5/6     | 41/43   |
| Python-Minifier                          | 13/13 | 24/24 | 6/6     | 43/43   |
| <i>Mean AUROC</i>                        |       |       |         |         |
| Random walk                              | 0.513 | 0.507 | 0.531   | 0.512   |
| w/o advanced                             | 0.529 | 0.525 | 0.544   | 0.529   |
| Python-Minifier                          | 0.506 | 0.505 | 0.519   | 0.507   |

Table 4: Watermark detectability under the random-walk obfuscator, its variant without control-flow flattening and opaque predicates (w/o advanced), and Python-Minifier, over the 43 effective Python settings (13 WLLM, 24 SWEET, 6 SynthID). The upper block counts settings whose post-obfuscation AUROC falls in the chance-level interval (0.4, 0.6); the lower block reports mean post-obfuscation AUROC.

**Results.** As shown in Table 4, the random-walk obfuscator drives all 43 settings into the chance-level interval (0.4, 0.6), with a mean AUROC of 0.512. The theoretically grounded construction therefore breaks the robustness of all three watermarking schemes in practice. Disabling control-flow flattening and opaque predicates leaves two settings outside the interval, with AUROCs of 0.61 and 0.64; the advanced transformations eliminate these remaining outliers and reduce the mean AUROC across all three schemes. Python-Minifier yields similarly chance-level AUROCs across the same 43 settings. Deterministic tools of this kind follow a fixed transformation path through the same rule-based transformation graph modeled by  $T$ ; the agreement between the randomized construction and Python-Minifier shows that these fixed paths

already realize, in practice, the limiting behavior that Theorem 3.3 establishes for the randomized walk.

## M Repository-Level Evaluation on SWE-bench Verified

The benchmarks in Subsection 4.1 consist of short, single-function programs. To examine whether our attack remains effective and affordable on longer, repository-level code generation, we evaluate it on SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024).

**Setup.** We randomly sample 50 tasks from the benchmark and solve them with mini-SWE-agent (Yang et al., 2024b; SWE-agent, 2025), using the official INT4-quantized Qwen3.5-35B-A3B (Qwen, 2026) as the backbone model. The non-watermarked agent achieves a 44% solve rate. We then apply WLLM under each of its 15 hyperparameter settings (the same grid as in Subsection 4.1), running the same 50 tasks once per setting, and apply the detector to the lines added by the agent’s patch. Watermark strength trades off sharply against task performance in this agentic setting: the two settings with the highest AUROCs, 0.750 and 0.774, achieve solve rates of only 26% and 16%, retaining at most 59.1% of the non-watermarked performance. For the seven quality-preserving settings that retain more than 80% of the non-watermarked solve rate, we use Python-Minifier to obfuscate the methods modified by the coding agent, reconstruct each patch, and re-evaluate the AUROCs.

| Metric      | Before obfuscation | After obfuscation |
|-------------|--------------------|-------------------|
| AUROC range | [0.518, 0.682]     | [0.432, 0.556]    |
| Mean AUROC  | 0.612              | 0.491             |

Table 5: WLLM detectability on SWE-bench Verified before and after obfuscation, over the seven settings that retain more than 80% of the non-watermarked solve rate.

**Results.** Table 5 reports detectability over the seven quality-preserving settings. Detectability is generally lower on repository-level tasks to begin with, and obfuscation further removes the remaining signal: all seven post-obfuscation AUROCs lie near 0.5. Increasing code and task complexity therefore does not restore watermark robustness. The attack also remains cheap at this scale: obfuscation takes 0.23 seconds per patch on average

using a single CPU core across the seven settings, and under SWE-bench’s official resource limits, no previously solved task times out or runs out of memory after obfuscation.

## N Detection Against Alternative Negative Classes

In Subsection 4.1, watermark detectability is measured by AUROC between the z-score distribution of the generated samples and the standard normal distribution. In deployment, a defender may instead calibrate the detector against concrete negative samples, such as non-watermarked model outputs, human-written code, or obfuscated non-watermarked code. We therefore recompute the AUROC scores for all WLLM Python experiments—the 15 hyperparameter settings on two models and two Python benchmarks, giving 60 experiments per negative class—against four such negative classes: non-watermarked LLM outputs; human-written code, taken as the canonical solutions of the benchmarks; and two classes of non-watermarked code obfuscated by each of the two off-the-shelf Python obfuscators, PyMinifier and Python-Minifier.

| Negative class     | Mean AUROC    | (.4, .6) | (.45, .55) |
|--------------------|---------------|----------|------------|
| Non-WM LLM code    | 0.800 → 0.518 | 60/60    | 52/60      |
| Human-written code | 0.805 → 0.518 | 60/60    | 49/60      |
| + PyMinifier       | 0.806 → 0.519 | 60/60    | 53/60      |
| + Python-Minifier  | 0.803 → 0.516 | 60/60    | 52/60      |
| Overall            | 0.804 → 0.518 | 240/240  | 206/240    |

Table 6: AUROC of all WLLM Python experiments recomputed against four negative classes. “Non-WM” denotes non-watermarked code; “+ PyMinifier” and “+ Python-Minifier” denote non-watermarked code obfuscated by the respective tool. Each row aggregates the 60 WLLM Python experiments (15 hyperparameter settings  $\times$  2 LLMs  $\times$  2 Python benchmarks); mean AUROC is reported before  $\rightarrow$  after obfuscation, and the counting columns refer to post-obfuscation scores.

Before obfuscation, WLLM remains detectable against every negative class. For example, with  $\delta = 3.0$  and  $\gamma = 0.25$ , the watermarked model retains 94.90% of the non-watermarked HumanEval Pass@1 while the AUROC is at least 0.937 for every negative class. After obfuscation, as Table 6 shows, the mean AUROC is approximately 0.518 for every negative class; all 240 AUROC scores fall within (0.4, 0.6), and 206 of 240 fall within (0.45, 0.55). The loss of watermark robustness under obfuscation is thus consistent across all four

negative classes, and our conclusion does not depend on the choice of the negative class.

## O Evaluation of ACW

Beyond the three N-gram-based schemes in our main experiments, we evaluate ACW (Guo and Cheng, 2025), a recent code watermarking scheme designed for practical and effective detection while preserving functionality.

Under its ACW-s setting, we generate 20 completions for each of the 164 HumanEval tasks, 3,280 programs in total, and obfuscate the watermarked outputs with Python-Minifier. After obfuscation, the detection AUROC drops to 52.92%, close to random guessing. The vulnerability to code obfuscation thus extends to this scheme as well, and we are not aware of current evidence that ACW is robust to code obfuscation.