

Robust and Secure Code Watermarking for Large Language Models via ML/Crypto Codesign

Ruisi Zhang ^{*1} Neusha Javidnia ^{*1} Nojan Sheybani ¹ Farinaz Koushanfar ¹

Abstract

This paper introduces RoSeMary, the first-of-its-kind ML/Crypto codesign watermarking framework that regulates LLM-generated code to avoid intellectual property rights violations and inappropriate misuse in software development. High-quality watermarks adhering to the detectability-fidelity-robustness tri-objective are limited due to codes' low-entropy nature. Watermark verification, however, often needs to reveal the signature and requires re-encoding new ones for code reuse, which potentially compromising the system's usability. To overcome these challenges, RoSeMary obtains high-quality watermarks by training the watermark insertion and extraction modules end-to-end to ensure (i) unaltered watermarked code functionality and (ii) enhanced detectability and robustness leveraging pre-trained CodeT5 as the insertion backbone to extract better code features. In the deployment, RoSeMary uses zero-knowledge proofs for secure verification without revealing the underlying signatures. Extensive evaluations demonstrated RoSeMary achieves high detection accuracy while preserving the code functionality. RoSeMary is also robust against attacks and provides efficient secure watermark verification.

1. Introduction

The AI-empowered code-generation LLMs, such as GitHub Copilot (GitHub, 2023), Qwen2.5-Coder (Hui et al., 2024), and Code LLaMA (Roziere et al., 2023), generate high-quality code via user instructions. They assist software engineers with agile development and reduce production costs (Tan et al., 2023; Cai et al., 2024). Developing such powerful models requires substantially more effort compared to natural languages, e.g., designing specialized tokenization modules (Li et al., 2022; Roziere et al., 2023) and

^{*}Equal contribution ¹University of California, San Diego. Correspondence to: Ruisi Zhang <ruz032@ucsd.edu>.

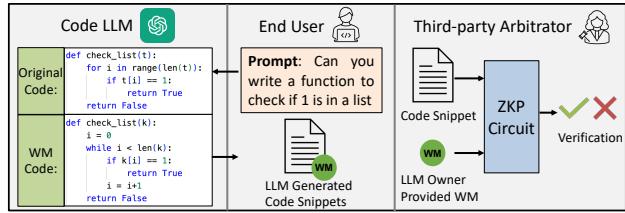


Figure 1. Overview of watermark insertion and extraction. The Code LLM owner watermarks the code before distributing the snippets to end users. The third-party arbitrator leverages zero-knowledge proofs to verify the ownership without requiring the owner to reveal the encoded watermark.

acquiring high-quality code training data (Lu et al., 2021; Puri et al., 2021). Nevertheless, AI-generated code may be used for malicious purposes and raise ethical and legal concerns, e.g., plagiarizing code that violates academic integrity (Cyphert, 2023; Tan et al., 2024) and contributing vulnerable code to open-source repositories (Panichella, 2024; Garg et al., 2024), etc.

Watermarking provides a promising solution to regulate LLM-generated content by embedding invisible signatures onto the code (Huo et al., 2024; Liu & Bu, 2024). Prior watermarking solutions fall into two approaches: (i) inference-based watermarking and (ii) neural-based watermarking. Inference-based watermarking (Lee et al., 2024; Ning et al., 2024) encodes watermarks by splitting vocabulary into green/red lists on high-entropy tokens and decoding the next token only from the green list. Such methods do not consider the syntactic constraints, which can corrupt the code functionality. Neural-based watermarking SrcMarker (Yang et al., 2024) employs a neural network to encode watermarks on both syntactic and variable name feature space for more robust watermarks. The shallow network architecture trained from scratch limits SrcMarker's code feature extraction ability to provide higher watermarking strength and results in reduced detectability.

Apart from the watermarking systems' detectability-fidelity-robustness imbalance, existing solutions face practical usability challenges. After disclosing the encoded signatures for third-party verification, code owners need to re-encode new ones to reuse the same code. Due to the code's low-entropy nature, high-quality watermarks that are detectable,

fidelity-preserving, and robust are limited. Encoding new signatures may corrupt the code’s usability.

RoSeMary leverages an ML/Crypto codesign approach to tackle these challenges and ground the usability of the code watermarking framework. It adopts the Seq-to-Seq CodeT5 (Wang et al., 2021) architecture, pre-trained on millions of high-quality code snippets, as the watermark insertion backbone to extract better code features to fuse with watermarks and improve watermark detectability. A transformer decoder is used for watermark extraction. The watermark encoder and decoder are trained end-to-end to (i) preserve functionality by minimizing the code feature loss between original and watermarked code after syntactic and variable rename transformations; and (ii) ensure detectability and robustness by minimizing the message extraction loss between the encoded signature and the extracted message from both the watermarked and the adversarially modified code. As such, RoSeMary strengthens the detectability-fidelity-robustness tri-objective for better watermarking performance.

As shown in Figure 1, the trained watermark encoder embeds the owner’s signature to the LLM-generated code and distributes the watermarked code to users. If a code snippet is suspected to be LLM-generated, users can submit the code to a third-party arbitrator for inspection and request the LLM owner input their signature to a zero-knowledge proof (ZKP) circuit for public verification. The ML/crypto codesign system enables efficient watermark source verification while keeping signatures private.

In brief, our contributions are summarized as follows:

- Developing an end-to-end code watermarking framework that balances the detectability-fidelity-robustness tri-objective for high-quality code watermarking.
- Leveraging the first-of-its-kind ML/Crypto codesign to enable secure watermark verification via zero-knowledge proofs. It verifies the code snippet source without revealing the encoded signatures.
- Performing evaluations on extensive code benchmarks, demonstrating RoSeMary (i) achieves 0.97 detection AUROC while preserving the code functionality and showing resilience against attacks and (ii) efficient in securely verifying the snippet within 120ms using zero-knowledge proofs.

2. Background and Related Work

Code Watermarking for Large Language Models Compared to natural language, watermarking code needs to preserve both its semantics and functionality. Prior work can be methodologically categorized into two approaches (Zhang

et al., 2024): (i) inference-based watermarking (Lee et al., 2024; Ning et al., 2024), and (ii) neural-based watermarking (Yang et al., 2024). The inference-based watermarking (Lee et al., 2024) encodes signatures at the LLM inference stage. It splits vocabulary into green/red lists only on high-entropy tokens and restricts the LLM decoding to predict the next token from the green list. However, such insertion loses the global view of the code, in which performing watermark insertions may violate syntactic constraints and corrupt code functionality. Neural-based code watermarking approach (Yang et al., 2024) tries to maintain code functionality by encoding watermarks on both the syntactic transformation structures and the variable names. It leverages a dual-channel neural network to embed watermarks on code feature space and decodes a set of probably over potential syntactic transformations, as well as the variable to rename. Nevertheless, SrcMarker (Yang et al., 2024) employs a shallow transformer trained from scratch for watermark insertion/extraction, which limits the code feature extractability and results in weak watermark detectability.

There is another line of work that employs rule-based methods (Li et al., 2023; 2024) to watermark code. It maintains a transformation table containing the transformation ID and the rule to transform the code. For each code segment, rule-based watermarking applies available transformations on the original code to form the watermark and obtains the watermarked snippets. The watermarks may be vulnerable to watermark removal attacks that statistically change the syntax. As such, we do not consider them in this paper.

Due to the code’s low-entropy nature, high-quality watermarks adhering to the detectability-fidelity-robustness tri-objectives are limited. After the watermark is revealed to the third party for legal verification, re-encoding another set of signatures on code data may hurt its usability. Prior solutions only design the code watermark insertion/detection algorithms without considering such cases for secure watermark verification to protect owner’s signatures.

Zero-knowledge Proofs (ZKPs) are a cryptographic primitive that allows a prover to prove knowledge of a secret value w to a verifier. In a standard ZKP scheme, the prover \mathcal{P} convinces a verifier \mathcal{V} that w is a valid private input such that $y = \mathcal{C}(x, w)$, in which \mathcal{C} is an arbitrary computation and x and y are public inputs and outputs, respectively. In general, ZKPs are extremely useful in computations where verification of outputs is costly (e.g. machine learning), as ZKPs allow users to verify a small proof rather than repeating the computation themselves (Xing et al., 2023). In most ZK schemes, the majority of the computation lies in the setup and proving phases, as any computation \mathcal{C} must be properly encoded in a way that ensures efficient processing during proof generation. In the context of ZK machine learning, for example, the layers, activation functions, and

parameters, must all be represented as *circuits*. This process, called *arithmetization*, generally involves the conversion of the computations into arithmetic operations that can be efficiently performed over a finite field (Mouris & Tsoutsos, 2021). The setup and arithmetization phases of ZKP protocols are typically where the cryptographic elements are injected to ensure the privacy of w .

Zero-knowledge proof generation can be performed in an interactive or non-interactive manner, depending on the application. One of the main drawbacks of interactive schemes is that they limit proofs to *designated-verifier* settings, meaning proof generation, which is the most computationally heavy process in ZKP workflows, must be repeated for every new verifier. Non-interactive ZKPs allow for the *publicly verifiable* setting, meaning that once a proof is generated attesting correct computation or valid data, it can be verified by any third party. Generally, non-interactive ZKP schemes can be represented with the three following algorithms:

- $(\mathcal{VK}, \mathcal{PK}) \leftarrow \text{Setup}(\mathcal{C})$: A trusted third party, when trusted setup is needed, or \mathcal{V} (with publicly verifiable randomness) runs a setup procedure to generate a prover key \mathcal{PK} and verifier key \mathcal{VK} .
- $\pi \leftarrow \text{Prove}(\mathcal{PK}, \mathcal{C}, x, y, w)$: \mathcal{P} generates proof π to convince \mathcal{V} that w is a valid witness. A malicious \mathcal{P} cannot generate a valid proof without knowledge of w . Alongside this, π does not reveal anything about w .
- $1/0 \leftarrow \text{Verify}(\mathcal{VK}, \mathcal{C}, x, y, \pi)$: \mathcal{V} accepts or rejects proof π . \mathcal{V} cannot be convinced by an invalid proof due to soundness property of ZKPs.

The most notable non-interactive ZKP scheme is Groth16-based zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs), which generate succinct proofs that are of constant size 128 bytes (Groth, 2016). Due to their succinctness, verification of zk-SNARKs is also very fast - in the order of milliseconds. The main drawback of zk-SNARKs that operate in the Groth16 proof system is the reliance on a computationally heavy trusted setup process, done by a trusted third party, in the presence of every new computation \mathcal{C} . This approach is best suited for applications in which \mathcal{C} is relatively static. ZKROWNN (Sheybani et al., 2023) shows the feasibility of Groth16 zk-SNARKs for watermark verification in deep neural networks (DNN), requiring only low communication and runtime for a user to verify a proof. However, its primary goal is to protect the watermarks of deep neural networks for IP protection of the models, rather than protecting the watermarks embedded in the data generated by a generative model, which is different from RoSeMary.

Although Groth16-based zk-SNARKs work well for computation on the scale of DNNs, their performance begins to

falter as \mathcal{C} grows, as they require quite heavy computation on the prover side to ensure succinctness. RoSeMary utilizes the Halo2 proof system (Zcash, 2024) to build efficient zk-SNARKs at a real-world scale, with support for dynamic \mathcal{C} . Halo2 utilizes a *universal* and *updatable* setup process, such that trusted setup does not have to be performed for every new \mathcal{C} . Besides this, Halo2 does not enforce constant size proofs. Instead, this proof system produces larger proofs, generally in the range of tens to hundreds of kilobytes, as a tradeoff to provide higher prover scalability.

3. Method

3.1. Threat Model

As shown in Figure 1, we aim to watermark LLM-generated code before distributing the content to users (Lee et al., 2024; Yang et al., 2024). The watermark insertion ensures the detectability of the encoded signature while maintaining code functionality unaltered and robustness against adversarial attacks. Due to the code’s low-entropy nature, high-quality watermarks aligning with those objectives are limited per code segment. Thus, we also aim to avoid revealing and re-encoding new signatures after the code source verification. We consider malicious end users may attempt to retain the code functionality but remove the encoded signature. The adversary has general knowledge of the watermarking framework, but he/she cannot access or manipulate the owner’s watermark insertion/extraction.

3.2. RoSeMary Design

RoSeMary consists of a watermark insertion and a watermark extraction module. As shown in Figure 2, the watermark insertion backbone \mathbf{S} takes the watermark message M and the code T as input and generates (i) a probability over the syntactic transformations and (ii) variable name distribution over the vocabulary. Then, the code is watermarked by performing the transformations to get the watermarked code $S(T, M)$. Then, a watermark decoder decodes the message M' from the watermarked code $S(T, M)$.

Watermark Insertion The watermark insertion employs the CodeT5 (Wang et al., 2021), pre-trained on millions of high-quality code files, as the backbone \mathbf{S} for watermark encoding. The encoder \mathbf{S}_e extracts the code feature and fuses with the message M ’s feature extracted by \mathbf{R}_m . The decoder \mathbf{S}_{d1} and \mathbf{S}_{d2} decodes two sets of probabilities over the syntactic transformations as p_{syn} and variable token distributions p_{var} . Then, RoSeMary obtains the watermarked code $S(T, M)$ by executing the predicted transformations from $\text{argmax}(p_{syn})$ and $\text{argmax}(p_{var})$. The syntactic transformation details are in Appendix B.

To mimic the malicious transformations the adversaries can perform over the watermarked code, the watermark inser-

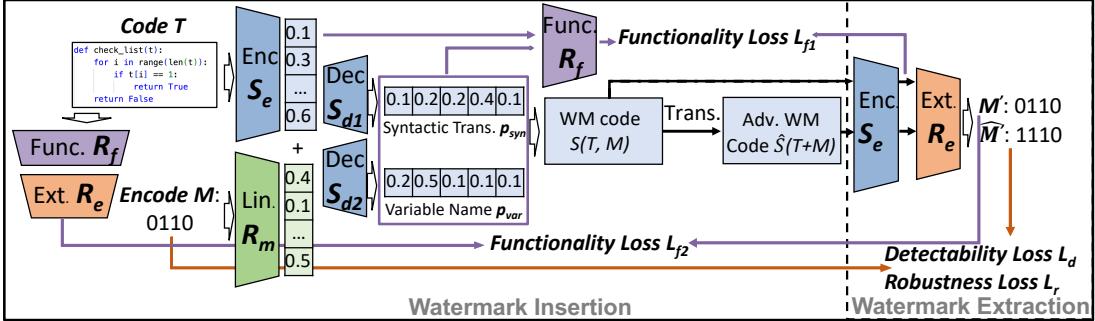


Figure 2. RoSeMary watermarking procedure. The watermark insertion takes the original code and watermark message as input and fuses their features by CodeT5’s encoder S_e . Two sets of decoders S_{d1} and S_{d2} predicts the probability over the available syntactic transformations and the renamed variable over the vocabulary. Then, the watermark extraction module decodes watermarks from the syntactic-transformed and variable-renamed watermarked code $S(T, M)$, as well as its malicious transformation $\hat{S}(T, M)$. The two parts are trained jointly to ensure (i) functionality-invariant by minimizing functionality loss L_f and (ii) accuracy and robust message decoding by minimizing detectability loss L_d and robustness loss L_r .

tion also perturbs the decoded probability p_{syn} and p_{var} . As shown in Equation 1, it adds Gaussian noise centered in 0 with variance equals σ_p . Then, RoSeMary obtains the adversarial example $\hat{S}(T, M)$ for robust message recovery during training.

$$\begin{aligned}\hat{p}_{syn} &= p_{syn} + \epsilon \\ \hat{p}_{var} &= p_{var} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_p^2)\end{aligned}\quad (1)$$

Watermark Extraction The watermark extraction decodes messages from the watermarked code $S(T, M)$. The encoder S_e , with shared parameters and architecture as the watermark insertion, is used to extract code features. Then, a shallow decoder R_e is used to recover message M' .

Watermarking Strength We measure the watermarking strength under the null hypothesis using z-score in Equation 2. The higher the z-score is, the more confident an owner can claim the code has been watermarked. M is a binary sequence whose message generation is random and follows binomial distributions. The probability for generating bit 0 is $p = 0.5$, and bit 1 is $1 - p = 0.5$. The mean of the message distribution is $\mu = |M| \times p$, and the variance can be calculated as $\sigma^2 = |M| \times p \times (1 - p)$. $|N|$ bits out of the message M match M' .

$$z = \frac{|N| - \mu}{\sigma} \quad (2)$$

3.3. RoSeMary End-to-End Training

We guide the training by minimizing the loss L in Equation 3. RoSeMary is trained to meet three criteria: (i) Functionality-invariant: the functionality of watermarked code $S(T, M)$ remains the same as the input code T as L_f ; (ii) Detectability: the decoded message M' matches the encoded M for successful detection as L_d ; (iii) Robust-

ness: the adversarial sample $\hat{S}(T, M)$ ’s decoded message \hat{M} matches the encoded M for robust detection as L_r .

$$L = w_f L_f + w_d L_d + w_r L_r \quad (3)$$

Functionality Loss Performing transformations over T results in non-differentiable watermarked $S(T, M)$. Inspired by SrcMarker (Yang et al., 2024), we employ R_f to approximate the watermark insertion/extraction. The R_f encourages the code functionality feature to be close during training. This is achieved by minimizing the mean square error (MSE) (Allen, 1971) L_{f1} between $R_f(S_e, p_{syn}, p_{var})$ and $S_e(S(T, M))$ as in Equation 4. We also ensure the R_f ’s approximation is correct for watermark extraction by minimizing the binary cross entropy (BCE) loss (Ruby & Yendapalli, 2020) L_{f2} between $R_f(T)$ and predicted M' .

$$L_f = MSE(R_f(S_e, p_{syn}, p_{var}), S_e(S(T, M))) + BCE(R_f(T), M') \quad (4)$$

Detectability Loss RoSeMary minimize the BCE loss between M and M' in Equation 5 to recover correct message in watermark extraction.

$$L_d = BCE(M, M') \quad (5)$$

Robustness Loss To enable robust message recovery over malicious transformations, the watermark extraction also decodes the malicious message \hat{M}' over $\hat{S}(T, M)$ and minimizes the BCE loss between M and \hat{M}' in Equation 6.

$$L_r = BCE(M, \hat{M}') \quad (6)$$

3.4. Secure Watermark Verification

The core problem with watermark extraction is the requirement of revealing the watermarked info to prove that you own something. This presents a significant challenge because each time a signature is exposed, the data must undergo re-watermarking to prevent adversaries from altering or erasing the exposed signature. Utilizing zero-knowledge proofs (ZKPs) we can solve this problem. We present a unique watermark extraction scheme, built using non-interactive ZKPs, that can efficiently prove that code has been generated using a proprietary LLM, without revealing what the original watermark was. Our solution generates publicly verifiable proofs, such that one proof can be generated and universally verified to prove that a code snippet was generated from a proprietary code LLM.

We utilize Halo2-based zk-SNARKs ([Zcash, 2024](#)), a class of non-interactive ZKPs that offer high scalability and fast verification time. Our proposed system benefits from the fact that proof generation only has to be done once, and, as proof generation is the slowest aspect of Halo2-based zk-SNARKs, we do not need to view this as a bottleneck. Due to the computational overhead of ZKPs, our approach includes non-interactive ZKP-specific optimizations, such as custom quantization, to ensure that the operation is runtime and memory-efficient.

A high-level approach towards our zero-knowledge watermark extraction scheme can be seen in Algorithm 1. To verify copyrights, the model owner starts by mapping the watermarked code $\mathcal{S}(T, M)$ into its embedding space using \mathbf{S}_e . This is done by running a feed-forward process on \mathbf{S}_e with input $\mathcal{S}(T, M)$, which results in a feature vector $\mathcal{S}(T, M)_{\text{embed}}$ that represents the watermarked code in the correct embedding space. Then, the model owner begins the ZKP generation process. $\mathcal{S}(T, M)_{\text{embed}}$ and a target bit error rate (BER) θ are taken in as public inputs, as they do not reveal any sensitive information about the proprietary LLM or watermarking scheme. The parameters of the shallow linear decoder \mathbf{R}_e and the original signature M are taken in as private inputs. With all computation represented as a zero-knowledge circuit, the trained watermark extraction module \mathbf{R}_e decodes the signature M' from the watermarked code by performing our custom *zkFeedForward* function on \mathbf{R}_e , with the input set to $\mathcal{S}(T, M)_{\text{embed}}$. This results in an extracted signature M' , of the same length as M . Within the same zero-knowledge circuit, the BER between the extracted signature M' and the original signature M that the LLM owner provides is calculated. This is done with our provided custom *zkBER* function, which returns 1 if the BER between M' and M is less than θ , or else it returns 0. The resulting proof π will only be valid if the extracted signature M' has a low enough BER compared to the original signature M . This proof π can be sent to any verifier \mathcal{V} to

prove that the inspected watermarked code was a result of the model owner’s proprietary LLM.

A majority of the computational burden lies in the *zkFeedForward*, which requires custom optimization of the shallow linear decoder to ensure efficient operation when translated to ZK computation. Specifically, \mathbf{R}_e is made up of batch normalization, fully-connected, ReLU, and dropout layers. To implement and run *zkFeedForward*, we use a customized version of the EZKL Rust package ([Zkonduit, 2024](#)). EZKL accepts a computational graph as input, allowing us to optimize our computation before converting it to the correct input format. We provide four custom optimizations and capabilities to ensure efficient proof generation, while maintaining small proof size and fast verification:

1. We lower the memory requirement that is necessary for non-linear layers by adding support for polynomial approximations, which is an important technique in privacy-preserving applications. We approximate ReLU using $\sigma(x) = x^2 + x$, which has been shown to closely replicate the ReLU ([Ali et al., 2020](#)).
2. We quantize parameters into Bfloat16 (BF16) format, a 16-bit floating point format that reduces the memory requirement for proof generation ([Burgess et al., 2019](#)), while maintaining network-level accuracy.
3. We add support for a highly efficient, zero-knowledge bit error rate calculation circuit based on the Halo2 proof system to represent *zkBER*. This is done using the bitwise AND operator to calculate the number of bits that differ between the M' and M .
4. We add a composability layer that allows for efficient combination of Halo2-based and EZKL circuits (e.g. *zkFeedForward* and *zkBER*) for representation in a single computational graph.

Using these optimizations, we are able to build an efficient ZK watermark extraction and verification scheme with small proofs and fast verification that cleanly integrates into RoSeMary’s end-to-end workflow.

Algorithm 1 ZK Watermark Extraction and Verification

- 1: **Public Values:** Watermarked text embedding $\mathcal{S}(T, M)_{\text{embed}}$, Target bit error rate (BER) θ
 - 2: **Private Input:** Shallow linear decoder \mathbf{R}_e , Signature M
 - 3: **Circuit:**
 - 4: $M' = \text{zkFeedForward}(\mathbf{R}_e)$ on input $\mathbf{S}_e(\mathcal{S}(T, M))$
 - 5: $\text{valid_BER} = \text{zkBER}(M, M', \theta)$
 - 6: **return** valid_BER
-

4. Experiment

We conduct comprehensive experiments to demonstrate:
(i) RoSeMary maintains balanced detectability-fidelity-

| Method | HUMAN EVAL (Chen et al., 2021b) | | | | MBPP (Austin et al., 2021) | | | | EVALPLUS (Liu et al., 2023) | | | |
|-------------------------|---------------------------------|-------------|-------------|-------------|----------------------------|-------------|-------------|-------------|-----------------------------|-------------|-------------|-------------|
| | Pass% | AUROC | TPR | FPR | Pass% | AUROC | TPR | FPR | Pass% | AUROC | TPR | FPR |
| Natural Language | | | | | | | | | | | | |
| KWG | 42.62% | 0.82 | 0.56 | 0.03 | 57.30% | 0.78 | 0.43 | 0.03 | 54.02% | 0.73 | 0.35 | 0.05 |
| REMARK-LLM | 0% | 0.97 | 0.88 | <u>0.04</u> | 0% | 0.98 | 0.89 | 0.05 | 0% | 0.98 | 0.96 | <u>0.05</u> |
| Code | | | | | | | | | | | | |
| SWEET | 82.53% | 0.87 | 0.59 | 0.02 | 90.00% | 0.86 | 0.47 | 0.05 | 84.90% | 0.86 | 0.52 | 0.04 |
| SrcMarker | <u>95.12%</u> | <u>0.90</u> | <u>0.76</u> | 0.07 | 97.95% | <u>0.91</u> | <u>0.76</u> | 0.06 | 95.57% | <u>0.92</u> | <u>0.81</u> | 0.07 |
| RoSeMary | 95.12% | 0.97 | 0.98 | 0.06 | <u>97.64%</u> | 0.97 | 0.99 | <u>0.05</u> | <u>95.39%</u> | 0.97 | 0.98 | 0.06 |

Table 1. RoSeMary performance on watermarking HumanEval (Chen et al., 2021b), MBPP (Austin et al., 2021), and DS-1000 (Lai et al., 2022) datasets when comparing with natural language watermarking KGW (Kirchenbauer et al., 2023a) and REMARK-LLM (Zhang et al., 2024); code watermarking SWEET (Lee et al., 2024) and SrcMarker (Yang et al., 2023). The best metric values are highlighted in **bold** text, the second best metric values are underlined, and grey means failed watermark insertion (0% pass rate).

robustness triangle in Section 4.2 and Section 4.4; (ii) RoSeMary incurs minimal secure watermark verification overhead via zero-knowledge proofs in Section 4.3.

4.1. Experiment Setup

Dataset and Evaluation Metrics We use HumanEval (Chen et al., 2021b), MBPP (Austin et al., 2021), and EvalPlus (including both HumanEval+ and MBPP+) (Liu et al., 2023) as the target benchmark to evaluate RoSeMary’s performance. All of the datasets have instruction prompts for code generation, human-written canonical solutions, and test cases for functionality evaluation.

We assess the watermarked code performance from the following aspects: (i) **Detectability**: classification metrics (AUROC for area under the receiver operating characteristic curve, TPR for true positive rates, and FPR for false position rates) over watermarked and non-watermarked codes’ z-score; (ii) **Fidelity**: the pass rate (Pass%) (Chen et al., 2021b) of watermarked code.

Baselines We compare RoSeMary with state-of-the-art natural language watermarking baselines: (1) **KGW** (Kirchenbauer et al., 2023a) is an inference-based watermarking scheme for natural language. It encodes watermarks at the LLM decoding stage by splitting the vocabulary into green/red lists and guides the decoding to primarily select tokens from the green list; (2) **REMARK-LLM** (Zhang et al., 2024) is a neural-based watermarking scheme for LLM-generated texts. We leverage CodeT5 as the watermark insertion backbone and take both the original code and watermarking signature as input. A watermark extraction module is used to decode the message, which is trained end-to-end with the insertion module to encourage close semantics and successful message extraction. State-of-the-art code watermarking baselines: (3) **SWEET** (Lee et al., 2024) is an inference-based watermarking scheme for LLM-generated code. It optimizes KGW’s decoding by setting an entropy threshold and encodes watermarks only toward high-entropy token decoding; (4) **SrcMarker** (Yang et al., 2024):

is a neural-based watermarking scheme for LLM-generated code. It employs shadow transformers for watermark insertion/extraction, where the watermark insertion generates the syntactic and variable rename transformation probabilities.

RoSeMary is pre-trained on CodeSearchNet (Husain et al., 2019), which collects the open-source non-fork repositories from GitHub and cleans the dataset for executable functions. For fair comparisons, we pre-train SrcMarker (Yang et al., 2024) and REMARK-LLM (Zhang et al., 2024) on the same dataset and report their respective detectability and fidelity performance. As SrcMarker does not support watermarking Python code, we train SrcMarker with the same Python syntactic transformations as RoSeMary in Appendix B. SWEET (Lee et al., 2024) encodes watermarks at the inference stage. We thus report Pass% of the watermarked code whose original one is functional. For both KGW (Kirchenbauer et al., 2023a) and SWEET (Lee et al., 2024), we use Qwen2.5-Coder-14B (Hui et al., 2024) as the code generation model.

Implementation Details We include more RoSeMary’s implementation details in Appendix D.

4.2. Watermark Detectability and Fidelity Performance

The watermarking performance of RoSeMary and baselines on HumanEval (Chen et al., 2021b), MBPP (Austin et al., 2021), and EvalPlus (Liu et al., 2023) in Table 1. We highlight RoSeMary is able to provide high detectability while maintaining code functionality invariant.

Compared to KGW (Kirchenbauer et al., 2023b) KGW watermarks LLM-generated code by promoting token decoding from the green list of the vocabulary. We relax the green/red list split ratio γ to 0.5 and set the constant δ added on green tokens’ probabilities to 3 to guide watermark insertion on green lists while ensuring Code LLM generates compilable code. However, as some of the low-entropy tokens are sensitive to alterations, restricting the watermark insertion results in an average of 48.69% pass rate drop to achieve an average of 0.78 AUROC for watermark detection.

Compared to REMARK-LLM (Zhang et al., 2024) REMARK-LLM primarily replaces words with their synonyms or changes the sentence syntax for watermark insertion. It was able to learn code semantics by leveraging CodeT5 (Wang et al., 2021) as the watermark insertion backbone and pre-training on large code datasets. However, it transforms code without syntactic constraints, making the watermarked code non-compilable. As such, while reaching significant detectability, REMARK-LLM has low pass rates that corrupted the watermarked codes’ functionality.

Compared to SWEET (Lee et al., 2024) SWEET improves over KGW by restricting watermark decoding on high-entropy tokens to maintain both detectability and fidelity. However, encoding watermarks at inference time loses the global view of the code and fewer syntactic transformations can be made to encode the watermark. As such, it weakens the watermarking strength and results in an average of 11.34% lower AUROC for watermark detection compared to RoSeMary. While SWEET avoids watermarking on high-entropy tokens, such watermark insertions without considering code syntactic constraints still result in 10.24% pass rate drop compared to RoSeMary.

Compared to SrcMarker (Yang et al., 2023) SrcMarker uses shallow transformers trained from scratch for watermark insertion, which limits its code feature extraction ability. RoSeMary, on the other hand, leverages CodeT5 (Wang et al., 2021) as the backbone for better watermark insertion and results in an average of 6.59% higher AUROC scores and 26.61% higher TPR among all benchmarks than SrcMarker. Besides, both RoSeMary and SrcMarker perform transformations adhering to syntactic constraints and watermarks code with less than 5% pass rate drop.

4.3. Zero Knowledge Watermark Verification Overhead

We benchmark our ZK watermark extraction and verification by describing algorithm 1 in a computational graph that can be easily translated into a zero-knowledge circuit. We highlight that our approach demonstrates virtually no loss in utility. The computation is done using the EZKL framework (Zkonduit, 2024) in conjunction with the Halo2 proof system (Zcash, 2024), resulting in a small zk-SNARK proof that any third-party arbitrator can easily verify. Our approach ensures that no information is leaked about the parameters of the linear decoder \mathbf{R}_e and the watermarking methodology, including the original watermark M . The verification overhead is in Table 2. Generating the zk-SNARK proof for a watermark of 4 bits only takes the prover **P 6.79 seconds**, while only requiring a maximum of approximately **2.79 GB** of RAM. While this is significantly slower than standard inference, we highlight that this is a fully privacy-preserving solution, and, more importantly, proof generation only has to be done once. This process results in a proof

π of size **18.75 KB**, which can be transmitted to as many verifiers as necessary. We benchmarked the verification over 25 examples. This proof can be verified by any verifier \mathcal{V} in an average of **120 milliseconds**, while only requiring a maximum of approximately **227.88 MB** of RAM. All the verifier needs to verify a proof is the proof π and the verifier key \mathcal{VK} , which is only **511 KB** in our setup. This results in a required communication cost of less than a megabyte. As the proof generation time is amortized due to it only being performed once, RoSeMary’s watermark extraction and verification scheme is an extremely communication and runtime-efficient solution that ensures the security of watermarks that are applied to LLM-generated code samples.

| | Frequency | Comm. Size | RAM | Time |
|------------------|-----------|--------------------------------|-----------|-------|
| Proof Generation | Once | Proof Size: 18.75 KB | 2.79 GB | 6.79s |
| WM Verification | Every WM | \mathcal{V} Key Size: 511 KB | 227.88 MB | 120ms |

Table 2. Zero-knowledge Watermark Verification Overhead

4.4. Robustness Evaluations

As in Section 3.1, we assume the adversary is an end-user leveraging LLM-generated code for malicious purposes. They avoid being caught the code is machine-generated by removing the encoded signatures. Following SWEET (Lee et al., 2024), we consider two attacks: (1) **Variable-rename Attack (VA)**: the adversary randomly renames the variable with another word of similar meaning from the vocabulary; (2) **Refactor Attack (RA)**: the adversary refactors the watermarked code with open-source code LLM. We instruct Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) to refactor the code. The attack performance is evaluated on the MBPP dataset (Austin et al., 2021) with results in Figure 3. We compare RoSeMary with code watermark baselines SrcMarker (Yang et al., 2023) and SWEET (Lee et al., 2024).

As seen, RoSeMary keeps over 0.93 AUROC under variable-rename attack even after 50% of the variable names are replaced, whereas SrcMarker and SWEET demonstrate 0.07 and 0.21 lower AUROC. Similarly, RoSeMary keeps 0.73 AUROC under refactor attack, demonstrating its resilience toward attacks. The robustness primarily comes from two aspects: (i) leveraging CodeT5 (Wang et al., 2021) with enhanced code feature extraction in \mathbf{S}_e enables robust message recovery and (ii) adversarial training helps the watermark extraction module learns potential malicious transformations and decodes accurate watermark signatures.

4.5. Ablation Study and Analysis

This subsection provides details of how different components would impact RoSeMary performance and analysis of RoSeMary’s capacities. More analysis is in Appendix A.

Impact of Training Loss Weights We analyze how different loss weight choices would impact the watermark

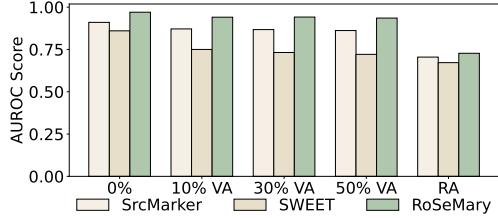


Figure 3. Robustness evaluation results under Variable-rename Attack (VA) and Refactor Attack (RA).

performance in Table 3. We pre-train RoSeMary on the CodeSearchNet (Husain et al., 2019) and evaluate the model performance on MBPP (Austin et al., 2021). As seen, when weighing more on w_d than w_f , RoSeMary provides higher AUROC and TPR metrics for verifying watermarked code. While a higher w_r results in better robustness against attacks, it results in slightly lower TPR in watermark detection.

| (w_f, w_d, w_r) | Pass% | AUROC | TPR | FPR |
|-------------------|---------------|-------------|-------------|-------------|
| (0.8, 0.1, 0.1) | 97.74% | 0.95 | 0.93 | 0.07 |
| (0.1, 0.8, 0.1) | 97.74% | 0.97 | 0.98 | 0.05 |
| (0.1, 0.1, 0.8) | 97.74% | 0.97 | 0.95 | 0.05 |

Table 3. Impact of RoSeMary’s training loss weights on the MBPP benchmark (Austin et al., 2021) performance.

Impact of Decoder S_{d1} and S_{d2} We analyze the impact of training with multiple decoders in Table 4. We pre-train RoSeMary on the CodeSearchNet (Husain et al., 2019), with either S_{d1} or S_{d2} , and evaluate the performance on MBPP (Austin et al., 2021). As seen, encoding watermarks with sole syntactic transformations (S_{d1}) or variable name transformations (S_{d2}) results in close pass rates but degraded detectability, as less information can be embedded onto the code for watermark insertion. Besides, encoding only syntactic transformations results in higher detectability. It primarily because such transformations carry more information for the watermark feature insertion.

| S_{d1} | S_{d2} | Pass% | AUROC | TPR | FPR |
|----------|----------|---------------|-------------|-------------|-------------|
| ✓ | ✗ | 98.05% | 0.81 | 0.36 | 0.06 |
| ✗ | ✓ | 98.77% | 0.73 | 0.24 | 0.06 |
| ✓ | ✓ | 98.15% | 0.97 | 0.99 | 0.07 |

Table 4. Impact of RoSeMary’s decoders on the MBPP benchmark (Austin et al., 2021) performance.

Watermarked Examples We show the watermarking examples in Table 4, in which the upper code is the original code and the lower one is the watermarked one. As seen, RoSeMary will transform the variable names and syntactic structure with meaningful content while maintaining the functionality invariant. For example, Listing 1’s two if statements (line 5 and line 6) are merged into one (line 5) in Listing 2. The addition statement (line 4 in Listing 1 and Listing

ing 2) is also changed from $_sum = _sum + arr[i]$ to $sum += arr[i]$. The variable name is updated from $_sum$ to sum . Additional examples are in Appendix C.

Listing 1. Original code

```

1 def check_last (arr,n,p):
2     _sum = 0
3     for i in range(n):
4         _sum = _sum + arr[i]
5         if p == 1:
6             if _sum % 2 == 0:
7                 return "ODD"
8             else:
9                 return "EVEN"
10            return "EVEN"
```

Listing 2. Watermarked code

```

1 def check_last (arr,n,onomies):
2     sum = 0
3     for i in range(n):
4         sum += arr[i]
5     if (onomies == 1 and sum % 2 == 0):
6         return "ODD"
7     return "EVEN"
```

Figure 4. Watermarked example randomly selected from HumanEval (Chen et al., 2021a). The upper code shows the original code and the lower code shows the watermarked code, where all watermarks are successfully extracted.

Watermark Insertion Overhead The time taken for watermark insertion is in Table 5, which is the average overhead for encoding signatures onto 50 examples from MBPP (Austin et al., 2021). RoSeMary embeds watermarks 90% faster than SWEET’s inference-based approach, which requires entropy calculation for every token and split vocabulary into green/red lists for high-entropy tokens. Compared to SrcMarker, while RoSeMary introduces more complex architectures, the additional watermark insertion overhead is less than 0.01s per sample. As such, RoSeMary’s watermark insertion is efficient.

| Method | SWEET | SrcMarker | RoSeMary |
|----------|-------|-----------|----------|
| Time (s) | 0.234 | 0.021 | 0.027 |

Table 5. Watermark insertion overhead for different code watermarking frameworks.

5. Conclusion

In this paper, we present RoSeMary, the first-of-its-kind ML/Crypto codesign secure watermarking framework with enhanced and balanced detectability-fidelity-robustness. We train the watermark insertion and extraction modules end-to-end, aiming to ensure the watermarked codes’ functionality-invariant, while maintaining the detectability of the watermark in the adversarial environment. We also design a zero-knowledge proof-based watermark verification in the system deployment to ensure correct ownership proofs without disclosing the signature details. Extensive evaluations of various coding benchmarks demonstrated the effectiveness of our proposed approach.

Impact Statement

Our code watermarking framework has potential societal implications. By building secure and public-verifiable code watermarking framework, our approach can enhance the usability of existing watermarking systems, especially for low-entropy code data. As such, it helps to detect code plagiarism for academic dishonesty, protect the intelligent property of the LLM owners, and monitor the distribution of the watermarked content. However, there might be cases where human-written code can be erroneously detected as LLM-generated, leading to false accusations.

References

- Ali, R. E., So, J., and Avestimehr, A. S. On polynomial approximations for privacy-preserving and verifiable relu networks. *arXiv preprint arXiv:2011.05530*, 2020.
- Allen, D. M. Mean square error of prediction as a criterion for selecting variables. *Technometrics*, 13(3):469–475, 1971.
- Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K., and Mansell, D. Bfloat16 processing for neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pp. 88–91. IEEE, 2019.
- Cai, Z., Chen, J., Chen, W., Wang, W., Zhu, X., and Ouyang, A. F-codellm: A federated learning framework for adapting large language models to practical software development. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pp. 416–417, 2024.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. 2021a.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.
- Cyphert, A. B. Generative ai, plagiarism, and copyright infringement in legal documents. *Minn. JL Sci. & Tech.*, 25:49, 2023.
- Garg, A., Degiovanni, R., Papadakis, M., and Le Traon, Y. On the coupling between vulnerabilities and llm-generated mutants: A study on vul4j dataset. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 305–316. IEEE, 2024.
- GitHub. GitHub Copilot. <https://docs.github.com/en/copilot>, 2023.
- Groth, J. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology—EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8–12, 2016, Proceedings, Part II 35*, pp. 305–326. Springer, 2016.
- Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Dang, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.
- Huo, M., Somayajula, S. A., Liang, Y., Zhang, R., Koushanfar, F., and Xie, P. Token-specific watermarking with enhanced detectability and semantic coherence for large language models. *ICML*, 2024.
- Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.
- Kirchenbauer, J., Geiping, J., Wen, Y., Katz, J., Miers, I., and Goldstein, T. A watermark for large language models. In *International Conference on Machine Learning*, pp. 17061–17084. PMLR, 2023a.
- Kirchenbauer, J., Geiping, J., Wen, Y., Shu, M., Saifullah, K., Kong, K., Fernando, K., Saha, A., Goldblum, M., and Goldstein, T. On the reliability of watermarks for large language models. *arXiv preprint arXiv:2306.04634*, 2023b.
- Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, W.-T., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.
- Lee, T., Hong, S., Ahn, J., Hong, I., Lee, H., Yun, S., Shin, J., and Kim, G. Who wrote this code? watermarking for code generation. *ACL*, 2024.

-
- Li, B., Zhang, M., Zhang, P., Sun, J., and Wang, X. Resilient watermarking for llm-generated codes. *arXiv preprint arXiv:2402.07518*, 2024.
- Li, Y., Choi, D., Chung, J., Kushman, N., Schrittweis, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alpha-code. *Science*, 378(6624):1092–1097, 2022.
- Li, Z., Wang, C., Wang, S., and Gao, C. Protecting intellectual property of large language model-based code generation apis via watermarks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2336–2350, 2023.
- Liu, J., Xia, C. S., Wang, Y., and Zhang, L. 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*, 2023. URL <https://openreview.net/forum?id=1qvx610Cu7>.
- Liu, Y. and Bu, Y. Adaptive text watermark for large language models. *ICML*, 2024.
- Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.
- Mouris, D. and Tsoutsos, N. G. Zilch: A framework for deploying transparent zero-knowledge proofs. *IEEE Transactions on Information Forensics and Security*, 16:3269–3284, 2021.
- Ning, K., Chen, J., Zhong, Q., Zhang, T., Wang, Y., Li, W., Zhang, Y., Zhang, W., and Zheng, Z. Mcgmark: An encodable and robust online watermark for llm-generated malicious code. *arXiv preprint arXiv:2408.01354*, 2024.
- Panichella, S. Vulnerabilities introduced by llms through code suggestions. In *Large Language Models in Cybersecurity: Threats, Exposure and Mitigation*, pp. 87–97. Springer Nature Switzerland Cham, 2024.
- Puri, R., Kung, D. S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.
- PyTorch Contributors. PyTorch. <https://pytorch.org/>, 2023. Last Access on December 26, 2022.
- Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- Ruby, U. and Yendapalli, V. Binary cross entropy with deep learning technique for image classification. *Int. J. Adv. Trends Comput. Sci. Eng.*, 9(10), 2020.
- Sheybani, N., Ghodsi, Z., Kapila, R., and Koushanfar, F. Zkrownn: Zero knowledge right of ownership for neural networks. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2023.
- Tan, C. W., Guo, S., Wong, M. F., and Hang, C. N. Copilot for xcode: Exploring ai-assisted programming by prompting cloud-based large language models. *arXiv preprint arXiv:2307.14349*, 2023.
- Tan, H., Duan, M., Liu, D., Zhou, L., Ren, A., Tan, Y., Zhong, K., et al. Rethinking literary plagiarism in llms through the lens of copyright laws. In *The 16th Asian Conference on Machine Learning (Conference Track)*, 2024.
- Wang, Y., Wang, W., Joty, S., and Hoi, S. C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.
- Xing, Z., Zhang, Z., Liu, J., Zhang, Z., Li, M., Zhu, L., and Russello, G. Zero-knowledge proof meets machine learning in verifiability: A survey. *arXiv preprint arXiv:2310.14848*, 2023.
- Yang, B., Li, W., Xiang, L., and Li, B. Towards code watermarking with dual-channel transformations. *arXiv preprint arXiv:2309.00860*, 2023.
- Yang, B., Li, W., Xiang, L., and Li, B. Srcmarker: Dual-channel source code watermarking via scalable code transformations. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 97–97. IEEE Computer Society, 2024.
- Zcash. Halo2 proof system. <https://github.com/zcash/halo2>, 2024.
- Zhang, R., Hussain, S. S., Neekhara, P., and Koushanfar, F. {REMARK-LLM}: A robust and efficient watermarking framework for generative large language models. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 1813–1830, 2024.
- Zkonduit. Ezkl. <https://github.com/zkonduit/ezkl>, 2024.

A. Additional Analysis

We analyze the effect of the message length on the watermarking performance in Table 6. As seen, increasing the message length results in degraded detectability, as the transformations to carry watermarks is limited on low entropy data. The code has to compromise the detectability to accommodate higher-bit watermarks.

| Message Length | Pass% | AUROC | TPR | FPR |
|----------------|--------|-------|------|------|
| 4 bit | 98.15% | 0.97 | 0.99 | 0.07 |
| 8 bit | 97.74% | 0.88 | 0.73 | 0.13 |

Table 6. Impact of message length on the MBPP benchmark (Austin et al., 2021) performance.

B. Available Syntactic Transformations

We provide details of the syntactic transformations used in RoSeMary in Table 7. The transformations include naming style, loop type, loop condition, nested conditions, operator substitution, and parentheses in conditions.

| Name | Description | Example |
|---------------------------|--|--|
| Naming Style | Choice of naming conventions: PascalCase, camelCase, snake_case, underscore_init, or ALL_CAPS. | <code>myVariable = 10 # camelCase my_variable = 10 # snake_case</code> |
| Loop Type | Choice between using a <code>for</code> loop or a <code>while</code> loop. | <code>for i in range(10): print(i)</code> <code>while i < 10: print(i)</code> |
| Loop Condition | Specification of conditions for infinite loops. | <code>while True: print("Running")</code> <code>while 1: print("Running")</code> |
| Nested Conditions | Use of merged conditions versus explicitly nested conditions. | <code>if x > 0 and y > 0: print("Both positive")</code> <code>if x > 0: if y > 0: print("Both positive")</code> |
| Operator Substitution | Choice between regular assignment and augmented assignment for binary operations. | <code>x = x * 5 # Regular x *= 5 # Augmented</code> |
| Parentheses in Conditions | Option to include or omit parentheses in <code>if</code> or <code>while</code> conditions. | <code>if (x > 0 and y < 10): print("Valid")</code> <code>if x > 0 and y < 10: print("Valid")</code> |

Table 7. Syntactic transformations provided by RoSeMary to watermark Python code.

C. Additional Watermarked Examples

We include additional watermarked examples in the Figure 5. The examples are randomly taken from HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021). All of the watermarks are successfully extracted. As seen, the watermarked code is functionality correct and the variables are replaced with close semantics.

Listing 3. Original code #1

```
def histogram(test):
    dict1={}
    list1=test.split(" ")
    t=0
    for i in list1:
        if(list1.count(i)>t) and i!="":
            t=list1.count(i)
    if t>0:
        for i in list1:
            if(list1.count(i)==t):
                dict1[i]=t
    return dict1
```

Listing 5. Original code #2

```
def get_closest_vowel(word):
    if len(word) < 3:
        return ""
    vowels = {"a", "e", "i", "o", "u", "A", "E", "O", "U", "I"}
    for i in range(len(word)-2, 0, -1):
        if word[i] in vowels:
            if (word[i+1] not in vowels) and (word[i-1] not in vowels):
                return word[i]
    return ""
```

Listing 7. Original code #3

```
def find_Extra(arr1,arr2,n) :
    for i in range(0, n) :
        if (arr1[i] != arr2[i]) :
            return i
    return n
```

Listing 9. Original code #4

```
def fizz_buzz(n: int):
    ns = []
    for i in range(n):
        if i % 11 == 0 or i % 13 == 0:
            ns.append(i)
    s = ''.join(list(map(str, ns)))
    ans = 0
    for c in s:
        ans += (c == '7')
    return ans
```

Listing 4. Watermarked code #1

```
def histogram(test):
    dict1={}
    list1=test.split(" ")
    t=0
    for item in list1:
        if ((list1.count(item)>t) and item!=""):
            t=list1.count(item)
    if (t>0):
        for item in list1:
            if(list1.count(item)==t):
                dict1[item]=t
    return dict1
```

Listing 6. Watermarked code #2

```
def getClosestVowel(dates):
    if (len(dates) < 3):
        return ""
    vowels = {"a", "e", "i", "o", "u", "A", "E", "O", "U", "I"}
    for i in range(len(dates)-2, 0, -1):
        if (dates[i] in vowels and (dates[i+1] not in vowels)
            and (dates[i-1] not in vowels)):
            return dates[i]
    return ""
```

Listing 8. Watermarked code #3

```
def find_extra(arr1,arr2,n) :
    for ena in range(0, n) :
        if arr1[ena] != arr2[ena] :
            return ena
    return n
```

Listing 10. Watermarked code #4

```
def FizzBuzz(n: int):
    Ns = []
    for I in range(n):
        if (I % 11 == 0 or I % 13 == 0):
            Ns.append(I)
    S = ''.join(list(map(str, Ns)))
    Ans = 0
    for C in S:
        Ans += (C == '7')
    return Ans
```

Figure 5. Watermarked example from HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021). The left code shows the original code and the right code shows the watermarked code, where all watermarks are successfully extracted.

D. RoSeMary’s Implementation Details

Hardware Infrastructure Our code is implemented using PyTorch (PyTorch Contributors, 2023). The training and inference of our watermarking models are performed on NVIDIA RTX A6000 GPUs with Ubuntu 20.04.5 LTS and Intel(R) Xeon(R) Gold 6338 CPU.

Implementation Details The training hyperparameters and model architecture settings are in Table 8.

| Training-time | Settings | SubModule | Backbones | Input Size | Output Size | SubModule | Backbones | Input Size | Output Size |
|--------------------------|-------------|------------------------|-----------|------------|-------------|--------------------|----------------|------------|-------------|
| Epoch, Batch size | 20, 16 | Feat. \mathbf{R}_f | Linear | 2304 | 768 | Enc \mathbf{S}_e | CodeT5 Encoder | 512 | 768 |
| w_f, w_d, w_r | 1, 1, 0.05 | Lin. \mathbf{R}_m | Linear | 4 | 768 | Ext \mathbf{R}_e | Linear | 768 | 4 |
| Maximum Token Size | 512 | Dec. \mathbf{S}_{d1} | Linear | 1536 | 320 | | | | |
| Optimizer, Learning rate | AdamW, 5e-5 | Dec. \mathbf{S}_{d2} | Linear | 1536 | 32100 | | | | |
| σ_p | 0.1 | | | | | | | | |

Table 8. RoSeMary’s implementation details. From left to right, we show training hyperparameters, watermark insertion architecture, and watermark extraction architecture details.