Large Language Models Are Effective Code Watermarkers

Anonymous ACL submission

Abstract

The widespread use of large language models (LLMs) and open-source code has raised ethical and security concerns regarding the distribution and attribution of source code, including unauthorized redistribution, license violations, and misuse of code for malicious purposes. Watermarking has emerged as a promising solution for source attribution, but existing techniques rely heavily on hand-crafted transformation rules, abstract syntax tree (AST) manipulation, or task-specific training, limiting their scalability and generality across languages. Moreover, their robustness against attacks remains limited. To address these limitations, we propose **CodeMark-LLM**, an LLM-driven watermarking framework that embeds watermark into source code without compromising its semantics or readability. CodeMark-LLM consists of two core components: (i) *Semantically Consistent Embedding* module that applies functionality-preserving transformations to encode watermark bits, and (ii) *Differential Comparison Extraction* module that identifies the applied transformations by comparing the original and watermarked code. Leveraging LLMs' cross-language generalization, CodeMark-LLM avoids language-specific engineering and training pipelines. Extensive experiments across diverse programming languages and attack scenarios demonstrate its robustness, effectiveness, and scalability.

1 Introduction

With the rapid development of the open-source ecosystem and the significant improvement of large language models' (LLMs) ability, LLMs have shown strong performance in tasks such as source code generation, refactoring and understanding (Shrivastava et al., 2023; Deng et al., 2023; Wei et al., 2023; Pei et al., 2023). However, the unauthorized use of source code has long posed a security risk—one that is becoming increasingly critical with the growth of developer communities and the rapid advancement of LLMs (Khoury et al., 2023; Liu et al., 2023). For instance, plagiarists may replicate open-source code with slight modifications and redistribute it under a different license to illegitimately claim ownership. In addition, LLMs may inadvertently reproduce copyright-protected code from their training data (Sun et al., 2022) or generate plausible but insecure code without clear attribution. These risks not only threaten the integrity of the software ecosystem but also underscore the urgent need for mechanisms that support code traceability and ownership verification. Therefore, designing an efficient and practical mechanism for source code traceability and ownership verification has become an urgent research challenge.

Currently, traceability and copyright protection technologies, mainly using digital watermarking schemes (Singh and Chadha, 2013), are widely used for images (Baluja, 2017; Hayes and Danezis, 2017), audio (Boney et al., 1996; Liu et al., 2024b) and text (Chang and Clark, 2014; Yang et al., 2022). The success of digital watermarking techniques in the multimedia domain is attributed to the inherent tolerance of human perception to minor alterations. However, such methods are not applicable to the source code domain, as even minor alterations can break syntactic correctness or change program behavior. Unlike multimedia data, source code is subject to strict syntactic rules and precisely defined semantics, and structural integrity must be maintained to ensure functional equivalence and compilability (Wan et al., 2022). In addition, the diversity of programming languages and differences in coding styles further limit the direct applicability of traditional watermarking methods in the software domain.

Recent approaches have made certain progress in addressing these challenges but still exhibit notable limitations. As shown in Table 1, these methods often suffer from several limitations: (i) high de-

Method	Training-Free	Automatic	Parser-Independent	Language-Agnostic	Robustness
CodeMark (Li et al., 2023)	✗	✗	✗	✗	✗
SrcMarker (Yang et al., 2024)	✗	✗	✗	✗	✓
ACW (Li et al., 2024)	✓	✗	✗	✗	✗
CodeIP (Guan et al., 2024)	✗	✗	✓	✓	✓
RoSeMary (Zhang et al., 2025)	✗	✗	✓	✓	✓
CodeMark-LLM	✓	✓	✓	✓	✓

Table 1: Qualitative comparison of code watermarking methods across five criteria: **Training-Free** (does not require model training), **Automatic** (requires no handcrafted rules), **Parser-Independent** (does not rely on AST or syntax tools), **Language-Agnostic** (supports multiple languages without language-specific design), and **Robustness** (resilient to code modifications). ✓: Supported; ✗: Not supported.

sign complexity: these methods (Li et al., 2023; Yang et al., 2024; Li et al., 2024; Guan et al., 2024; Zhang et al., 2025) require handcrafted transformation rules, AST-based code rewriting, or additional model training, which reduces scalability and deployment flexibility. (ii) lack language generality: their designs rely heavily on language-specific features and cannot be directly applied across multiple programming languages. (iii) limited robustness: CodeMark (Li et al., 2023) and ACW (Li et al., 2024) are vulnerable to common attacks.

To address the above limitations, we propose CodeMark-LLM, an LLM-based source code watermarking framework that opens up a new paradigm in code watermarking research, which comprises two main components: Semantically Consistent Embedding and Differential Comparison Extraction. *In the watermark embedding phase*, Semantically Consistent Embedding leverages prompt-driven LLM to automatically generate and apply semantic-preserving code transformations, eliminating reliance on training, handcrafted rules, or AST parsing. These transformations are adaptively selected to support diverse programming styles, enabling broad language generality. *In the watermark extraction phase*, Differential Comparison Extraction performs multi-granularity comparisons between candidate and watermarked code to identify applied transformations and decode the watermark. This differential analysis, grounded in LLM's code reasoning capabilities, ensures robust watermark recovery even under common obfuscation or reformatting attacks.

To substantiate the efficacy of our proposed method, we conducted comprehensive experiments across datasets encompassing multiple programming languages, including C, C++, Java, and JavaScript. The results show that, compared with other methods, the proposed approach achieves strong stealthiness and efficiency while maintaining high embedding capacity and the watermarked code can pass the syntax check and unit tests with nearly 100% ratio. These findings highlight the great potential of LLMs to provide efficient, scalable, and robust solutions for code watermarking. The main contributions of this work are as follows:

- We propose **CodeMark-LLM**, a training-free code watermarking framework that eliminates reliance on handcrafted rules, AST tools, or model fine-tuning, addressing key design bottlenecks in prior work.
- We introduce a modular design comprising *Semantically Consistent Embedding* and *Differential Comparison Extraction*, enabling automatic transformation and resilient recovery without language-specific customization.
- We conduct extensive experiments across multiple programming languages and attack scenarios, demonstrating that CodeMark-LLM achieves superior fidelity, robustness, and language generality compared to prior approaches.

2 Related Work

Software watermarking aims to embed watermarks into software as a proof of ownership (Dey et al., 2019). It can be further divided into static and dynamic watermarking. *Static watermarking* embeds watermarks by directly modifying code structures or binaries (Balachandran et al., 2014; Chen et al., 2018; Collberg and Sahoo, 2005). For example, Kang et al. (Kang et al., 2021) modified binary function order to encode watermarks, while Monden et al. (Monden et al., 2000) added virtual methods to Java code for embedding bit strings. However, these methods focus on compiled binaries or intermediate representations, limiting their applicability to raw source code.

Dynamic watermarking encodes watermarks into the runtime behavior of programs, typically by modifying control flow or inserting special runtime states (Chen et al., 2017; Ma et al., 2019; Tian et al., 2015; Wang et al., 2018). Although dynamic watermarking can be robust in some settings, it requires the execution of software, making it impractical for

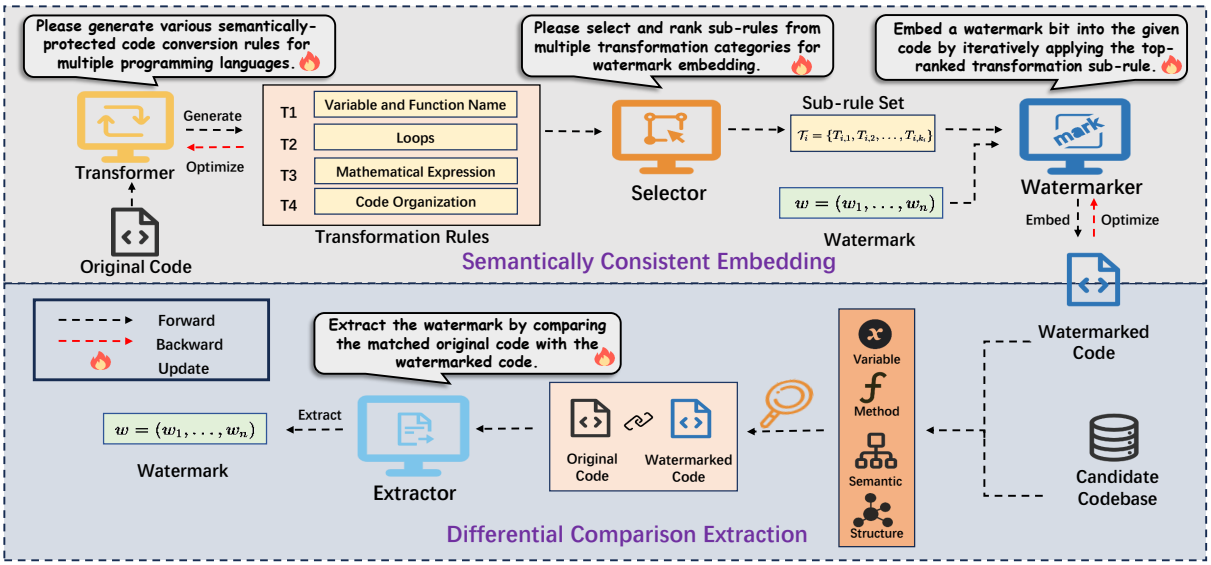


Figure 1: The overall framework of CodeMark-LLM.

source code snippets or lightweight development environments. Moreover, both static and dynamic schemes are rule-based and manually built, limiting scalability and cross-language generalization.

Semantic-preserving watermarking techniques address limitations of traditional approaches by enabling flexible watermark embedding while maintaining program functionality (Quiring et al., 2019; Yang et al., 2022; Zhang et al., 2020). RopGen (Li et al., 2022) designed 23 handcrafted transformation rules for C, C++, and Java. NatGen (Chakraborty et al., 2022) further leveraged these rules to pretrain LLMs for semantic understanding. CodeMark (Li et al., 2023) employed a GNN-based variable renaming strategy to preserve readability, yet suffers from limited capacity and vulnerability to renaming attacks. SrcMarker (Yang et al., 2024) combines rule-based transformations with neural models, but depends on handcrafted rules, AST parsing, and expensive training, risking syntax errors and requiring dedicated decoders. ACW (Li et al., 2024) uses fixed, manually defined code transformations to embed watermarks, limiting adaptability across languages and reducing robustness in diverse code contexts.

The emergence of LLMs such as GPT (OpenAI, 2022) and DeepSeek (Liu et al., 2024a) has enabled new possibilities in code generation and transformation (Jiang et al., 2024). With prompt engineering, LLMs have been applied to areas such as code completion (Zhang et al., 2024), translation (Jana et al., 2024), data augmentation (Ding et al., 2024) and text steganography (Wu et al., 2024). Inspired by the above, we propose CodeMark-LLM, a novel code watermarking framework that utilizes LLM to dynamically select semantic-preserving transformations. It enables broader transformations and better cross-language scalability.

3 Method

This section presents the design of the CodeMark-LLM framework, which consists of two core modules: Semantically Consistent Embedding and the Differential Comparison Extraction. Each module leverages LLM for language-agnostic, semantics-preserving watermarking. The overall workflow is illustrated in Figure 1.

3.1 Preliminaries and Problem Formulation

Task Overview. We study the problem of embedding watermark into source code while preserving its functionality, syntax validity, and readability. This facilitates ownership verification and source tracing in open-source ecosystems. We leverage LLM to automatically generate, apply, and reason over semantic-preserving code transformations. This LLM-centric formulation allows CodeMark-LLM to operate across programming languages with minimal human intervention.

Notation. Let $\mathcal{C}_0$ be the set of valid, compilable source code, and $\mathcal{C}_1$ the set of watermarked code. A code snippet is denoted by $\boldsymbol{c}_0 \in \mathcal{C}_0$ (original code) or $\boldsymbol{c}_1 \in \mathcal{C}_1$ (watermarked code). The watermark is a binary sequence $w = (w_1, \ldots, w_n) \in \mathcal{W}$, where $n = 4$ and $\mathcal{W}$ is the set of n-bit sequences. The embedding process is then modeled as:

$$\boldsymbol{c}_1 = \text{Embed}(\boldsymbol{c}_0, w), \quad (1)$$

where $\boldsymbol{c}_1$ preserves the functionality, syntax validity, and readability of $\boldsymbol{c}_0$. We focus on traceability and ownership verification, where the defender maintains a candidate codebase $\mathcal{D}$ that may con-

Category	Sub-Rule Type	Example (Original → Transformed)
Variable / Function Name	CamelCase to snake_case	`testStream()` → `test_stream()`
	snake_case to CamelCase	`my_var` → `myVar`
	To PascalCase	`remove()` → `Remove()`
	To UPPERCASE	`value` → `VALUE`
	To lowercase	`Value` → `value`
	Add suffix	`data` → `dataVal`
Loops	for to while	`for(...)` → `while(...){...}`
	while to for	`while(...)` → `for(...)`
	Flatten nested loop	`for(i){...for(j)...}` → `for(k)...`
	while to do-while	`while(c){...}` → `do{...} while(c);`
	Step increment	`i++` → `i+=1`
	Reverse loop	`for(i=0;...)` → `for(i=n-1;...)`
Math Expression	Group ops	`x + y + z` → `x + (y + z)`
	Mul to add	`2 * x` → `x + x`
	Factorization	`a*b + a*c` → `a*(b + c)`
	Identity transform	`x*x - y*y` → `(x - y)*(x + y)`
	Div to reciprocal	`a / b` → `a * (1 / b)`
	Pow to mul	`x*x` → `x * x`
	Expand distributive	`a*(b + c)` → `a*b + a*c`
	Sqrt-square	`sqrt(x*x)` → `abs(x)`
Code Organization	Optimize cond.	`if(x>0) return;` → `if(!(x>0)) return;`
	Reorder decl.	`int a; str b;` → `str b; int a;`
	Swap params	`f(a, b)` → `f(b, a)`
	Format spacing	`int x=5;` → `int x = 5;`
	Add braces	`if(a) return;` → `if(a){ return; }`
	Reorder cond.	`if(a && b)` → `if(b && a)`
	Insert blank line	`x=1; y=2;` → `x=1; \\y=2;`
	Adjust op space	`x=y+z;` → `x = y + z;`
	Inline temp var	`int t=x+y; return t;` → `return x+y;`
	Split decl.	`int x=0,y=1;` → `int x=0; int y=1;`

Table 2: CodeMark-LLM's set of transformation rules.

tain the original code. The closest match $\hat{\boldsymbol{c}} \in \mathcal{D}$ is identified, and the watermark is recovered via:

$$\hat{w} = \text{Extract}(\boldsymbol{c}_1, \hat{\boldsymbol{c}}), \tag{2}$$

where $\hat{w}$ is the final watermark.

3.2 Semantically Consistent Embedding

Transformation Rule Set Design. The transformation rule set $\mathcal{T}$ encodes watermark information into semantic-preserving code transformations, forming the core of watermark embedding. To address the limitations of static, language-specific transformation methods, CodeMark-LLM employs LLM to construct a diverse, cross-language rule set $\mathcal{T}$:

$$\mathcal{T} = \bigcup_{i=1}^{m} \mathcal{T}_i, \quad \mathcal{T}_i = \{T_{i,1},\ T_{i,2},\ \ldots,\ T_{i,k_i}\} \tag{3}$$

where $m = 4$ is the total number of transformation classes, and k_i is the number of sub-rules in the i-th class $\mathcal{T}_i$, with each $T_{i,j}$ representing a specific semantic-preserving transformation. We generate $\mathcal{T}$ once with an LLM and then fix it for all embedding and extraction; Table 2 reports the taxonomy and representative examples. All transformations preserve the semantics of the original code. As a result, we construct a set of sub-rules spanning four transformation classes, where sub-rules are instantiated by the LLM within the predefined semantic-preserving transformation categories, as summarized in Table 2. This expressive and diverse rule set forms the foundation for flexible and resilient bit-wise watermark encoding in source code.

Contextual Rule Selection with LLM Assistance. To support bit-wise watermark embedding under syntactic and semantic constraints, CodeMark-LLM employs a two-stage transformation strategy guided by LLMs. Given $\boldsymbol{c}_0$ and w, the LLM first analyzes the code structure and determines applicable transformation types. For each selected type, LLM ranks candidate rules by suitability to the input code, producing a prioritized sequence of sub-rules represented as $\mathcal{L}_k(\boldsymbol{c}_0) = [T_{k,1}, T_{k,2}, \ldots]$ for each transformation class $\mathcal{T}_k$. To further enhance stealthiness and reduce the risk of pattern leakage, we apply a category balancing strategy that evenly distributes transformation types across the watermark bits, thereby improving robustness against statistical detection.

Watermark Embedding Execution. Given $\boldsymbol{c}_0$, w, and the sub-rule ranking lists $\mathcal{L}_k(\boldsymbol{c}_0) = [T_{k,1}, T_{k,2}, \ldots] \subseteq \mathcal{T}_k$ for each bit position k, the watermark embedding process proceeds iteratively as follows:

$$\boldsymbol{c}^{(k)} = \begin{cases} T_k^*(\boldsymbol{c}^{(k-1)}), & \text{if } w_k = 1 \\ \boldsymbol{c}^{(k-1)}, & \text{if } w_k = 0 \end{cases}, \qquad \boldsymbol{c}^{(0)} = \boldsymbol{c}_0, \quad \boldsymbol{c}_1 = \boldsymbol{c}^{(n)} \tag{4}$$

Dataset	Method	BitAcc (%)	MsgAcc (%)	BPF
CSN-Java	AWT_{code}	93.91	78.41	4
	SrcMarker	97.26	92.74	4
	CodeMark-LLM	**98.05**	**95.79**	4
CSN-JS	AWT_{code}	89.33	63.97	4
	SrcMarker	96.34	89.84	4
	CodeMark-LLM	**98.08**	**95.62**	4
GH-C	AWT_{code}	95.10	81.70	4
	$\text{CALS}_{\text{code}}$	96.07	92.81	1.22
	SrcMarker	93.36	79.52	4
	CodeMark-LLM	**97.49**	**92.81**	4
GH-Java	AWT_{code}	95.05	82.40	4
	$\text{CALS}_{\text{code}}$	94.43	91.83	1.40
	SrcMarker	90.93	75.14	4
	CodeMark-LLM	**97.05**	**92.74**	4

Table 3: The success rate of watermark extraction for different watermarking methods and the embedding capacity (BPF).

where T_k^* is the highest-ranked sub-rule selected from $\mathcal{L}_k(\boldsymbol{c}_0)$ for embedding the k-th bit w_k.

To ensure valid watermark embedding, each transformation T_k^* is selected from a predefined semantic-preserving rule set. If the top-ranked transformation is unsuitable for the code context, the LLM selects the next feasible candidate from the ranked list $\mathcal{L}_k(\boldsymbol{c}_0)$ to generate $\boldsymbol{c}^{(k)}$. This bounded retry improves embedding success while preserving watermark accuracy and stealth.

3.3 Differential Comparison Extraction

Multi-Feature Matching Retrieval. Given $\boldsymbol{c}_1$ and $\mathcal{D}$, CodeMark-LLM retrieves the most probable original code $\hat{\boldsymbol{c}}$ via a joint similarity function:

$$\hat{\boldsymbol{c}} = \arg\max_{\boldsymbol{c}_i \in \mathcal{D}} \Big[\alpha\, \text{Sim}_m(\boldsymbol{c}_i, \boldsymbol{c}_1) + \beta\, \text{Sim}_v(\boldsymbol{c}_i, \boldsymbol{c}_1) + \gamma\, \text{Sim}_s(\boldsymbol{c}_i, \boldsymbol{c}_1) + \delta\, \text{Sim}_{\text{sem}}(\boldsymbol{c}_i, \boldsymbol{c}_1)\Big], \tag{5}$$

$$\text{Sim}_m(\boldsymbol{c}_i, \boldsymbol{c}_1) = 1 - \frac{\text{LevDist}(\text{name}(\boldsymbol{c}_i), \text{name}(\boldsymbol{c}_1))}{\max\left(|\text{name}(\boldsymbol{c}_i)|, |\text{name}(\boldsymbol{c}_1)|\right)}, \tag{6}$$

$$\text{Sim}_v(\boldsymbol{c}_i, \boldsymbol{c}_1) = \frac{|V(\boldsymbol{c}_i) \cap V(\boldsymbol{c}_1)|}{|V(\boldsymbol{c}_i) \cup V(\boldsymbol{c}_1)|}, \tag{7}$$

$$\text{Sim}_s(\boldsymbol{c}_i, \boldsymbol{c}_1) = \cos\left(\vec{f}_s(\boldsymbol{c}_i), \vec{f}_s(\boldsymbol{c}_1)\right), \tag{8}$$

$$\text{Sim}_{\text{sem}}(\boldsymbol{c}_i, \boldsymbol{c}_1) = 1 - \frac{\text{LevDist}(\text{norm}(\boldsymbol{c}_i), \text{norm}(\boldsymbol{c}_1))}{\max\left(|\text{norm}(\boldsymbol{c}_i)|, |\text{norm}(\boldsymbol{c}_1)|\right)}, \tag{9}$$

where Sim_m, Sim_v, Sim_s, and Sim_{sem} denote similarity scores on method signatures, variable usage, structural, and semantic features, respectively. The $\text{name}(\cdot)$ denotes the function name, $V(\cdot)$ is the set of variable identifiers, $\vec{f}_s(\cdot)$ is a vector of structural token counts, and $\text{norm}(\cdot)$ is the normalized function string after whitespace removal. We set $\alpha = \beta = \gamma = \delta = 0.25$ in all experiments. The parameter selection is determined via grid search and detailed in Appendix C.5.

Rule Inference and Watermark Recovery. For each watermark bit w_k, LLM plays a pivotal role by reconstructing the guided sub-rule list $\mathcal{L}_k(\hat{\boldsymbol{c}}) = [T_{k,1}, T_{k,2}, \ldots] \subseteq \mathcal{T}_k$ from the matched original code $\hat{\boldsymbol{c}}$, adapting the process from Section 3.2 using contextual understanding. The LLM selects the top-ranked sub-rule $T_k^* = \mathcal{L}_k(\hat{\boldsymbol{c}})[0]$ based on its analysis of syntactic and structural features. We recover the bit using:

$$\hat{w}_k = \begin{cases} 1, & \text{if } T_k^*(\hat{\boldsymbol{c}}) \approx \boldsymbol{c}_1 \\ 0, & \text{otherwise} \end{cases}, \tag{10}$$

where $\approx$ indicates that applying the candidate sub-rule T_k^* to the original code $\hat{\boldsymbol{c}}$ produces the observed watermarked code $\boldsymbol{c}_1$ while preserving syntax validity and functionality. If the initial match is unclear, the LLM iteratively considers alternative sub-rule selections to improve accuracy and robustness. The final watermark is $\hat{w} = [\hat{w}_1, \ldots, \hat{w}_n]$.

4 Evaluation

In this section, we evaluate CodeMark-LLM. We first describe the experimental setup in Section 4.1. For CodeMark-LLM, we evaluate its watermark accuracy (Section 4.2), transparency (Section 4.3), efficiency and economic cost (Section 4.4) and robustness (Section 4.5).

4.1 Experiment Setup

Datasets and Preprocessing. To evaluate CodeMark-LLM across languages, we use three dataset types for C, C++, Java, JavaScript, and Python. For CSN-Java and CSN-JS (Husain et al.,

Method	Metric	Syntax		Execution			
		CSN-Java	CSN-JS	MBCPP	MBJP	MBJSP	MBPP
AWT_{code}	BitAcc(%)	93.91	89.33	97.12	93.88	83.97	/
	Pass(%)	0.18	0.51	0.00	0.00	0.00	/
$CALS_{code}$	BitAcc(%)	-	-	92.89	93.31	93.50	/
	Pass(%)	-	-	68.19	68.65	76.77	/
SrcMarker	BitAcc(%)	97.26	96.34	96.04	99.44	96.94	/
	Pass(%)	93.09	**100**	97.64	97.86	97.99	/
CodeMark-LLM	BitAcc(%)	**98.05**	**98.08**	**99.64**	**99.72**	**99.47**	**97.85**
	Pass(%)	**99.85**	99.11	**99.35**	**99.31**	**99.87**	**99.69**

Table 4: Operational semantic results based on performed operations. For CSN, we use syntax checking; for MBXP, we use execution-based checking. "**-**" indicates that the method was not evaluated on the dataset due to prohibitive computational cost. "/" indicates that the method cannot be applied to the Python language.

2019), functions are paired with natural language descriptions from open-source projects. GitHub-C and GitHub-Java are used for C/C++ and Java in smaller-scale evaluation under baseline limits. For execution-based validation, we use MBXP (Athiwaratkun et al., 2022) datasets, including C++, Java, JavaScript and Python (MBCPP, MBJP, MBJSP, MBPP). Each function embeds a 4-bit watermark. Detailed settings are in Appendix C.1.

Baselines and LLMs. We choose Srcmarker, AWT_{code} and $CALS_{code}$ proposed in Srcmarker as the baselines. AWT_{code} and $CALS_{code}$ are obtained by modifying AWT (Abdelnabi and Fritz, 2021) and CALS (Yang et al., 2022), both of which are natural language watermarking tools. AWT_{code} shares a similar architecture with AWT but uses source code datasets for training. $CALS_{code}$ replaces the original BERT (Devlin et al., 2019) with CodeBERT (Feng et al., 2020) to better accommodate source code data.For LLM-based code watermarking, we use the GPT-4o API for embedding and extraction; results with DeepSeek-V3 and Gemini 1.5 Pro are reported in Appendix C.

Threat models. We consider two attacker models: random removal (the attacker is unaware of the embedding pipeline details) and adaptive dewatermarking (the attacker knows watermarking uses four transformation categories but not the specific sub-rules); details are in Appendix C.2.

4.2 Watermark Accuracy

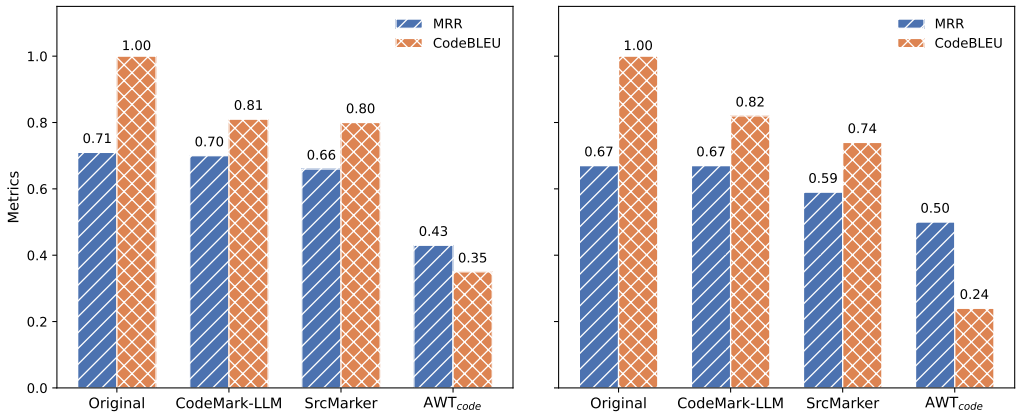
Metrics. We compare CodeMark-LLM with the baselines on two types of datasets for accuracy and capacity. Accuracy is measured by Bit Accuracy (BitAcc) and Message Accuracy (MsgAcc). BitAcc denotes the percentage of correctly extracted bits, while MsgAcc represents the percentage of entire messages that are correctly recovered. Capacity is measured in terms of the average number of bits in the embedding function (BPF).

Results. Table 3 shows that CodeMark-LLM outperforms all baselines across datasets. Compared to SrcMarker, AWT_{code}, and $CALS_{code}$, CodeMark-LLM achieves a significant breakthrough in deployment efficiency by eliminating the need for training, while maintaining high embedding capacity (BPF = 4). We study how extraction accuracy varies under different watermark capacities and report the results in Appendix C.7. $CALS_{code}$ faces efficiency bottlenecks, preventing deployment on CSN-JS (3.5k) and CSN-Java (10k) datasets. To assess cross-model robustness, we conduct experiments where the watermarker and extractor use different LLMs, and report the results in Appendix C.8.

4.3 Transparency

Metrics. We evaluate operational semantics using syntax checking and execution-based tests to verify functional correctness of watermarked code. As functions in the CSN dataset are not compilable independently, we use `tree-sitter` to identify AST errors. For MBXP, we employ the pass rate—the fraction of watermarked code passing all unit tests. Natural semantics assessment uses CodeBLEU (Ren et al., 2020) and MRR. CodeBLEU evaluates similarity through syntax, data flow, and n-gram matching, measuring differences following CodeXGLUE (Lu et al., 2021). MRR evaluates watermarking's effect on code naturalness by measuring the retrieval rank of watermarked code for a natural language query, with a higher score indicating better semantic retention. The computation uses a fine-tuned CodeBERT (Feng et al., 2020).

Operational semantic results. The operational semantic results are displayed in Table 4. Training-based methods (AWT_{code}, $CALS_{code}$, SrcMarker) have limited generalizability: they cannot be applied to Python, while CodeMark-LLM naturally supports it and maintains strong performance. AWT_{code}, though trained on source code, performs



(a) Natural semantic evaluation results on CSN-JS

(b) Natural semantic evaluation results on CSN-JS

Figure 2: Natural semantics metrics for CodeMark-LLM, SrcMarker and AWT_{code}. "Original" refers to the unwatermarked code.

Method	Training Time (h)	Embedding Time (s)	Extraction Time (s)	Total Time (h)	Economic Cost ($)
AWT_{code}	61.50	0.1055	**0.0023**	61.50	0.0123
SrcMarker	13.32	**0.0741**	0.0034	13.32	0.0027
CodeMark-LLM	**0**	3.3333	1.3333	**5.85**	**0.0020**

Table 5: Comparison of runtime and per-sample economic cost across different watermarking methods.

poorly in execution-based evaluation, failing to preserve functionality. $\text{CALS}_{\text{code}}$, despite replacing BERT with CodeBERT, still fails over 25% of tests due to limited understanding of complex code rules. SrcMarker approaches CodeMark-LLM but relies on handcrafted transformations and AST rewriting, causing inconsistent variable renaming and failures in syntax or unit tests. Failure cases for the other methods are provided in Appendix C.4. CodeMark-LLM achieves nearly 100% BitAcc and Pass across datasets, retaining the original code semantics. The typical transformation example is in Appendix C.3.

Natural semantic results. Figure 2 illustrates the results for natural semantics. Higher MRR and CodeBLEU values indicate better preservation of natural semantics. On the CSN-JS dataset, CodeMark-LLM achieves results close to the original text in MRR, with only a 0.01 drop in CSN-Java, and maintains CodeBLEU scores above 0.81, demonstrating effective naturalness preservation. In contrast, both SrcMarker and AWT_{code} perform worse than CodeMark-LLM in preserving natural semantics. SrcMarker shows a larger decrease in MRR and CodeBLEU scores on both datasets, indicating limitations in naturalness preservation. AWT_{code} performs even worse, with CodeBLEU sharply reduced by syntax errors despite retaining searchable tokens and identifiers.

4.4 Efficiency and Economic Cost

Metrics. We compare CodeMark-LLM with the baselines using Per-sample Cost, which includes both total time (embedding and extraction time) and economic cost. SrcMarker and AWT_{code} require significant computational resources, so we estimate training costs using a common market rental price of $2/hour for GPU usage. Due to the extremely high runtime of $\text{CALS}_{\text{code}}$ (over 342 hours), we do not include it in the comparison.

Results. Table 5 shows that CodeMark-LLM achieves lower per-sample cost and shorter runtime than training-based methods. We report LLM inference calls per sample for embedding and extraction in Appendix C.10. While methods such as SrcMarker and AWT_{code} incur overhead from model training, their cross-language generalization is nearly zero because they require language-specific adapters for each target language. This limits their efficiency in large-scale multilingual deployment. By contrast, CodeMark-LLM requires no training, adapts naturally to different programming languages, and thus offers higher deployment flexibility and clear advantages in scalability.

4.5 Robustness

Random removal attack. We evaluate watermark robustness against a removal attack, where the adversary is aware of the use of code transformations for embedding but lacks knowledge of the exact transformation rules. Therefore, the most straightforward attempt to remove the watermark is to randomly perform variable renaming or code transformations. To simulate this, we randomly rename 25%, 50%, 75%, and 100% of variables, and apply

Attack	SrcMarker				CodeMark-LLM			
	GH-Java		GH-C		GH-Java		GH-C	
	BitAcc	CB	BitAcc	CB	BitAcc	CB	BitAcc	CB
No Atk.	90.93	-	93.36	-	**97.05**	-	**97.49**	-
T@1	78.81	45.34	89.49	42.89	**93.65**	**62.83**	**92.97**	**67.11**
T@2	73.46	45.12	81.59	42.81	**92.97**	**62.75**	**89.11**	**61.74**
T@3	69.83	44.53	79.08	42.75	**92.83**	**62.17**	**87.20**	**60.83**
V@25%	79.90	43.93	84.80	42.61	**96.42**	**66.24**	**96.13**	**69.32**
V@50%	78.58	43.78	82.24	42.48	**96.10**	**59.04**	**95.97**	**63.45**
V@75%	70.64	43.60	70.48	42.28	**95.28**	**51.91**	**95.70**	**56.36**
V@100%	59.80	43.42	62.69	41.98	**95.05**	**43.48**	**94.71**	**49.08**

Table 6: Performance under random removal attack. CB: CodeBLEU; T: random code transformation; V: random variable substitution.

Method	Metrics	MBJP	MBJSP	MBCPP	MBPP
SrcMarker	BitAcc (%)	50.12 (49.32↓)	50.72 (46.22↓)	49.18 (46.86↓)	/
	Pass (%)	98.46	99.25	99.74	/
	CodeBLEU (%)	**48.80**	**49.80**	**50.75**	/
CodeMark-LLM	BitAcc (%)	**76.93** (22.79↓)	**79.0**2 (20.45↓)	**83.48** (16.16↓)	**79.72**(18.13↓)
	Pass (%)	**99.17**	**99.87**	**99.74**	**99.38**
	CodeBLEU (%)	47.85	48.55	44.88	**41.19**

Table 7: Performance under adaptive de-watermarking. "Pass" is the percentage of de-watermarked code that passes the test case. Values in parentheses report the absolute decrease compared to the watermarked code before the attack."/" indicates that the method cannot be applied to the Python language.

up to 1, 2, or 3 random code transformations per snippet. We measure post-attack watermark recovery accuracy to assess robustness, and report CodeBLEU scores before and after the attack to quantify semantic preservation and perceptual cost. As shown in Table 6, CodeMark-LLM achieves high BitAcc across all attack intensities, demonstrating strong resilience to both structural and identifier-level perturbations. In contrast, SrcMarker shows a sharp decline in BitAcc under increasing attack strength, particularly under full variable renaming (V@100%), where accuracy drops to 59.80% on GH-Java and 62.69% on GH-C. This indicates its limited capacity to trace watermarks once identifiers are obfuscated. CodeMark-LLM's robustness stems from its rule-aligned embedding, which binds each watermark bit to a semantic transformation context and employs a multi-feature matching mechanism that enables accurate recovery even after aggressive code modifications. The results of random removal attacks on the MBXP dataset are provided in Appendix C.11.

Adaptive de-watermarking. We further simulate a stronger adversary who is aware of the LLM-based embedding mechanism but lacks access to the specific embedding strategy. To remove potential watermarks, the attacker paraphrases the watermarked code using a general-purpose LLM (GPT-4o), while preserving functional correctness. As shown in Table 7, the code retains a high execution success rate while exhibiting a clear drop in CodeBLEU, indicating effective code rewriting without altering functionality. Despite such high-level semantic changes, CodeMark-LLM consistently achieves high BitAcc, demonstrating strong resilience. CodeMark-LLM's watermarking is guided by semantically consistent transformation patterns, whose categories remain invariant even when low-level lexical variations occur. This allows reliable extraction by comparing the attacked code with the original and detecting transformation traces aligned with the predefined stylistic families.

5 Conclusion

We present CodeMark-LLM, a novel LLM-based framework that redefines the paradigm of source code watermarking. Unlike prior methods that rely on handcrafted rules, AST manipulations, or retraining, CodeMark-LLM uses prompt-driven semantic transformations to embed and extract watermarks without any model fine-tuning or language-specific engineering. This enables lightweight deployment and seamless adaptation to multiple programming languages. Extensive experiments demonstrate that CodeMark-LLM achieves strong robustness against common and adaptive attacks while maintaining high watermark fidelity and code functionality. These results highlight the potential of LLMs as a practical and generalizable foundation for secure and language-agnostic software ownership verification.

Limitations

While CodeMark-LLM demonstrates strong performance across multiple criteria, a few limitations remain. First, the use of LLMs introduces some non-determinism in generation, which may occasionally cause minor deviations such as formatting inconsistencies. These cases are rare and typically resolved via simple retry or prompt refinement. Second, although CodeMark-LLM is training-free, the reliance on commercial LLM APIs may introduce modest inference latency or cost in large-scale applications. Finally, our current implementation focuses on function-level watermarking; extending it to module-level or cross-file granularity remains a promising direction for future work.

References

Sahar Abdelnabi and Mario Fritz. 2021. Adversarial watermarking transformer: Towards tracing text provenance with data hiding. In *2021 IEEE Symposium on Security and Privacy*, pages 121–140.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, and Mingyue Shang. 2022. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*.

Vivek Balachandran, Ng Wee Keong, and Sabu Emmanuel. 2014. Function level control flow obfuscation for software security. In *Proceedings of the Eighth International Conference on Complex, Intelligent and Software Intensive Systems*, pages 133–140.

Shumeet Baluja. 2017. Hiding images in plain sight: Deep steganography. In *Advances in Neural Information Processing Systems*.

Laurence Boney, Ahmed H. Tewfik, and Khaled N. Hamdy. 1996. Digital watermarks for audio signals. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*, pages 473–480.

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

Ching-Yun Chang and Stephen Clark. 2014. Practical linguistic steganography using contextual synonym substitution and a novel vertex coding method. *Computational Linguistics*, 40(2):403–448.

Jianping Chen, Kui Li, Wanzhi Wen, Weixu Chen, and Chenxue Yan. 2018. Software watermarking for java program based on method name encoding. In *Proceedings of the International Conference on Advanced Intelligent Systems and Informatics 2017*, pages 865–874.

Zhe Chen, Chunfu Jia, and Donghui Xu. 2017. Hidden path: Dynamic software watermarking based on control flow obfuscation. In *2017 IEEE International Conference on Computational Science and Engineering and IEEE International Conference on Embedded and Ubiquitous Computing*, pages 443–450.

Christian Collberg and Tapas Ranjan Sahoo. 2005. Software watermarking in the frequency domain: Implementation, analysis, and attacks. *Journal of Computer Security*, 13(5):721–755.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 423–435.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186.

Ayan Dey, Sukriti Bhattacharya, and Nabendu Chaki. 2019. Software watermarking: Progress and challenges. *INAE Letters*, 4:65–75.

Bosheng Ding, Chengwei Qin, Ruochen Zhao, Tianze Luo, Xinze Li, Guizhen Chen, Wenhan Xia, Junjie Hu, Luu Anh Tuan, and Shafiq Joty. 2024. Data augmentation using llms: Data perspectives, learning paradigms and challenges. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 1679–1705.

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

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.

Jamie Hayes and George Danezis. 2017. Generating steganographic images via adversarial training. In *Advances in Neural Information Processing Systems*.

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

Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution. In *ECAI 2024*, pages 4011–4018. IOS Press.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint*, arXiv:2406.00515.

Honggoo Kang, Yonghwi Kwon, Sangjin Lee, and Hyungjoon Koo. 2021. Softmark: Software watermarking via a binary function relocation. In *Annual Computer Security Applications Conference*, pages 169–181.

Raphael Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt? *arXiv preprint*, arXiv:2304.09655.

Boquan Li, Mengdi Zhang, Peixin Zhang, Jun Sun, Xingmei Wang, and Zirui Fu. 2024. Acw: Enhancing traceability of ai-generated codes based on watermarking. *arXiv preprint*, arXiv:2402.07518.

Wei Li, Borui Yang, Yujie Sun, Suyu Chen, Ziyun Song, Liyao Xiang, Xinbing Wang, and Chenghu Zhou. 2023. Towards tracing code provenance with code watermarking. *arXiv preprint*, arXiv:2305.12461.

Zhen Li, Guenevere Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. Ropgen: Towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1906–1918.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint*, arXiv:2412.19437.

Hongbin Liu, Moyang Guo, Zhengyuan Jiang, Lun Wang, and Neil Gong. 2024b. Audiomarkbench: Benchmarking robustness of audio watermarking. *Advances in Neural Information Processing Systems*, 37:52241–52265.

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. *arXiv preprint*, arXiv:2305.01210.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Haoyu Ma, Chunfu Jia, Shijia Li, Wantong Zheng, and Dinghao Wu. 2019. Xmark: Dynamic software watermarking using collatz conjecture. *IEEE Transactions on Information Forensics and Security*, 14(11):2859–2874.

A. Monden, H. Iida, K. Matsumoto, K. Inoue, and K. Torii. 2000. A practical method for watermarking java programs. In *Proceedings 24th Annual International Computer Software and Applications Conference*, pages 191–197.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. OpenAI Blog.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *Proceedings of the International Conference on Machine Learning*, pages 27496–27520.

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

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: A method for automatic evaluation of code synthesis. *arXiv preprint*, arXiv:2009.10297.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *Proceedings of the International Conference on Machine Learning*, pages 31693–31715.

Prabhishek Singh and Ramneet Singh Chadha. 2013. A survey of digital watermarking techniques, applications and attacks. *International Journal of Engineering and Innovative Technology*, 2(9):165–175.

Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. In *Proceedings of the ACM Web Conference 2022*, pages 652–660.

Zhenzhou Tian, Qinghua Zheng, Ting Liu, Ming Fan, Eryue Zhuang, and Zijiang Yang. 2015. Software plagiarism detection with birthmarks based on dynamic key instruction sequences. *IEEE Transactions on Software Engineering*, 41(12):1217–1235.

Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. You see what i want you to see: Poisoning vulnerabilities in neural code search. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1233–1245.

Yilong Wang, Daofu Gong, Bin Lu, Fei Xiang, and Fenlin Liu. 2018. Exception handling-based dynamic software watermarking. *IEEE Access*, 6:8882–8889.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184.

Jiaxuan Wu, Zhengxian Wu, Yiming Xue, Juan Wen, and Wanli Peng. 2024. Generative text steganography with large language model. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 10345–10353.

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

Xi Yang, Jie Zhang, Kejiang Chen, Weiming Zhang, Zehua Ma, Feng Wang, and Nenghai Yu. 2022. Tracing text provenance via context-aware lexical substitution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 11613–11621.

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1169–1176.

Mingxuan Zhang, Bo Yuan, Hanzhe Li, and Kangming Xu. 2024. Llm-cloud complete: Leveraging cloud computing for efficient large language model-based code completion. *Journal of Artificial Intelligence General Science*, 5(1):295–326.

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

A Prompt of CodeMark-LLM

This section describes in detail the design of hints in CodeMark-LLM.

A.1 Prompt Design of Semantically Consistent Embedding

We provide the prompt template used by CodeMark-LLM to guide the LLM in performing semantically consistent watermark embedding. The prompt instructs the model to apply one transformation per bit based on the specified sub-rule list while preserving program semantics. The full structure is shown in Figure 3.

A.2 Prompt Design of Differential Comparison Extraction

To extract the watermark, CodeMark-LLM employs a comparison-based prompt that guides the LLM to identify applied transformations between original and watermarked code. This enables accurate bit recovery based on transformation traces. The prompt structure is shown in Figure 4.

B LLM Usage in CodeMark-LLM

In CodeMark-LLM, the LLM is used in both embedding and extraction. During embedding, the LLM (i) ranks candidate sub-rules in the finite list $\mathcal{L}_k(\boldsymbol{c}_0)$ for each transformation category, and (ii) instantiates the selected sub-rule on the input code. During extraction, given the matched original code $\hat{\boldsymbol{c}}$, the LLM reconstructs and ranks candidate sub-rules as $\mathcal{L}_k(\hat{\boldsymbol{c}})$ and supports checking which sub-rule best matches the observed watermarked code. In all cases, the candidates are restricted to the predefined semantic-preserving rule inventory (Table 2), and the rule inventory is not expanded during embedding or extraction.

C Additional Experimental Results

C.1 Dataset Preprocessing Steps

We processed the datasets following the data preprocessing methods outlined in the SrcMarker. Subsequently, the datasets were partitioned into training, validation, and test sets. For the GitHub-C and GitHub-Java datasets, we performed a random split with an 8:1:1 ratio. For the CSN dataset, we utilized its original train/valid/test split, while for the MBXP dataset, all samples were employed as the test set. The statistics of the datasets are shown in Table 8.

C.2 Threat Models

We consider two attacker models with different capabilities in our evaluation.

(1) Random removal attack. The attacker is unaware of the watermark embedding pipeline and can only apply unguided surface-level or structural perturbations, including variable renaming, loop form changes, and statement reordering through semantics-preserving transformations.

(2) Adaptive de-watermarking. The attacker fully knows that watermark embedding is implemented via four categories of semantics-preserving transformations, but does not know the specific sub-rule inventory. The attacker directly leverages **GPT-4o**

Prompt Template for Watermark Embedding in CodeMark-LLM

Role: You are a skilled programmer capable of rewriting code while preserving its functionality and syntax correctness.
Description: Given a source code and a randomly generated watermark bitstring, apply semantic-preserving transformations to embed the watermark.
Rules:

- For each bit $w_k = 1$, apply exactly one transformation from the k-th selected sub-rule.
- If $w_k = 0$, leave the code unchanged.
- Follow transformation-specific constraints.
- No functionality changes or added comments are allowed.

Example: Input: $w = (1, 0, 0, 1)$ and sub-rules {CamelCase to snake_case, for to while, group ops , insert blank line}
Output: Transformed code applying rules 1 and 4, skipping 2 and 3.
Task: Given code, bitstring $w = (w_1, \ldots, w_n)$, and sub-rule list {T}, output the watermarked code.

Figure 3: Prompt template for watermark embedding in CodeMark-LLM.

Prompt Template for Watermark Extraction in CodeMark-LLM

Role: You are an expert in identifying subtle transformations between source code versions.
Description: Given an original and a watermarked code, determine which transformations occurred to recover the embedded watermark.
Rules:

- For each bit position k, compare original and transformed code using sub-rule T_k.
- Set $w_k = 1$ if the rule was applied, otherwise $w_k = 0$.
- Justify decisions based on code structure and semantic consistency.

Example: Input: Code pair {Original, Watermarked}, sub-rules {T}
Output: $w = (1, 0, 0, 1)$ with justification per bit.
Task: Recover $w = (w_1, \ldots, w_n)$ from comparison between `[ORIGINAL]` and `[TRANSFORMED]` based on sub-rule sequence.

Figure 4: Prompt template for watermark extraction in CodeMark-LLM.

Dataset	GitHub		MBXP				CSN	
	C	Java	C++	Java	JS	Python	Java	JS
#Functions	4,577	5,501	764	842	797	974	173,326	63,258

Table 8: Dataset Statistics.

```java
public static int calc_sum(int a, int
    b) {
    int result = 0;
    for (int i = 0; i < 3; i++) {
        result = result + a + b;
    }
    return result;
}
```

(a) Original Code

```java
// method name renaming
public static int calcSum(int a, int b) {
    int result = 0;
    int i = 0;
    // loop transformed
    while (i < 3) {
        // expression reordered
        result = b + a + result;
        i++;
    }
    // return wrapped in block
    { return result; }
}
```

(b) Watermarked Code

Figure 5: A code snippet watermarked by CodeMark-LLM.

to perform global semantics-preserving paraphrasing of the code in order to maximally disrupt the embedded watermark signals. This attack simultaneously alters naming, code structure, control flow, and expression organization, and represents a stronger, targeted adaptive attack.

Defender capability. The defender has access to a fixed transformation rule set, the candidate codebase, and a deterministic bit-recovery procedure.

C.3 Transformation Examples for CodeMark-LLM

Fig. 5 shows the segments before and after code watermark embedding. The method name on line 1 has been changed from calc_sum to calcSum, which is a legal method name. The for loop on line 3 has been equivalently replaced by a while loop, introducing the additional variable i for control and adding i++ at the end of the loop. The expression in line 4 has been reordered and does not affect the functionality of the code. The return statement on line 6 is wrapped in a block of code. The transformation introduces no functional difference, demonstrating the effectiveness of CodeMark-LLM-based transformations.

C.4 Failure Cases of AWT_{code} , $\text{CALS}_{\text{code}}$, and SrcMarker

Three representative failure cases from **AWT_{code}**, **$\text{CALS}_{\text{code}}$**, and **SrcMarker** are shown in Figure 6, Figure 7, and Figure 8, respectively. In the case of AWT_{code}, the transformation introduces multiple syntax-breaking modifications. Specifically, it erroneously inserts spaces within function names and parameter lists, replaces valid constructor calls with malformed expressions, and introduces unmatched parentheses and invalid logical operators. For $\text{CALS}_{\text{code}}$, the transformation causes a typographical error in a variable name, changing `testTuple` to `testTuuple`. This syntactic mistake results in an undefined variable reference, which leads to a compilation failure. SrcMarker introduces both syntactic and semantic errors during watermark embedding. First, the original function parameter `onHotUpdateSuccess` is incorrectly replaced with `sizeMap`, while the internal logic still invokes `onHotUpdateSuccess()`, resulting in an undefined function call. Second, the error message string is corrupted by the removal of whitespace, significantly reducing its readability. Finally, the condition `typeof sizeMap = == 'function'` is invalid due to an incorrect equality check. These issues highlight SrcMarker's lack of context awareness and the absence of syntactic correctness verification in its transformation process.

C.5 Hyperparameter Selection

In Section 3.3, we define a joint similarity function (Eq. 5) that combines four similarity components, namely Sim_m, Sim_v, Sim_s, and Sim_{sem}, using the parameters α, β, γ, and δ. These parameters control the relative contribution of each similarity term in retrieving the most probable original code. To determine appropriate values for α, β, γ, and δ, we perform a grid search over the discrete set $\{0, 0.25, 0.5, 1\}$ for each parameter, subject to the constraint $\alpha + \beta + \gamma + \delta = 1$. All valid parameter combinations are evaluated on the MBJP dataset using BitAcc. Table 9 reports representative results. Among all evaluated configurations, the uniform setting $\alpha = \beta = \gamma = \delta = 0.25$ achieves the highest BitAcc of 99.72%. Accordingly, we adopt this setting for the joint similarity function in all experiments.

C.6 Generalization Across LLMs

To further validate the generalization ability of CodeMark-LLM, we conducted supplementary experiments on different LLMs. In addition to GPT-4o, we selected **DeepSeek-V3** and **Gemini 1.5 Pro** for evaluation. Under the same experimental setup, we compared the performance of watermark

```javascript
function createInstance(defaultConfig) {
  var context = new Axios(defaultConfig);
  var instance = bind(Axios.prototype.request, context);
  utils.extend(instance, Axios.prototype, context);
  utils.extend(instance, context);
  return instance;
}
```

(a) Original Code

```javascript
// Function name and parameter name split with spaces
function create Instance (default Config ) {
    // Invalid constructor and variable declaration
    var context , Math Axios (default Config );
    // Incorrect method call with mismatched brackets
    var instance , bind (Blog )prototype )request , context );
    // Logical operator and comment confusion
    // Copy raised to instance utils .extend (instance , Math .prototype , context )&&
    // Copy context to instance utils .extend (instance , context )&&
    &return instance &&
}
```

(b) Watermarked Code

Figure 6: Fail case of AWT_{code}. "Original" refers to the unwatermarked code.

α	β	γ	δ	BitAcc (%)
1	0	0	0	30.68
0	1	0	0	30.70
0	0	1	0	30.68
0	0	0	1	8.30
0.5	0.5	0	0	35.38
0.5	0	0.5	0	35.54
0.5	0	0	0.5	31.53
0	0.5	0.5	0	34.84
0	0.5	0	0.5	30.99
0	0	0.5	0.5	30.73
0.25	**0.25**	**0.25**	**0.25**	**99.72**

Table 9: Grid search results for the parameters α, β, γ, and δ on the MBJP dataset.

embedding and extraction, reporting two metrics: BitAcc and Pass. As shown in Table 10, CodeMark-LLM consistently achieves high performance on both models, with BitAcc remaining above 97% and Pass approaching 100%. This indicates that CodeMark-LLM does not rely on a specific LLM. Moreover, no systematic degradation was observed due to model differences, which provides strong evidence of the cross-model generalization capability of CodeMark-LLM.

C.7 Capacity

We analyze how watermark capacity affects extraction accuracy by varying the total number of embedded bits. Specifically, we evaluate watermark capacities of 2, 4, 6, and 8 bits and report BitAcc for CodeMark-LLM and the baselines. In CodeMark-LLM, each watermark bit is associated with one category of semantics-preserving transformations. Therefore, increasing the watermark capacity requires expanding the number of transformation categories and the corresponding sub-rule inventory. For higher-capacity settings, we extend the rule set accordingly while keeping the overall embedding and extraction pipeline unchanged. Table 11 reports the results on the MBJP dataset. As expected, all methods exhibit a gradual decrease in accuracy as the watermark capacity increases. However, CodeMark-LLM consistently achieves the highest BitAcc across all capacity settings. In particular, when embedding 8 bits, AWT_{code} and SrcMarker drop to 61.67% and 81.22%, respectively, whereas CodeMark-LLM still maintains a BitAcc of 95.10%. These results demonstrate that CodeMark-LLM remains robust under rule set expansion and that increasing watermark capacity does not significantly compromise extraction reliability.

C.8 Cross-LLM Embedding and Extraction

To evaluate the stability of CodeMark-LLM across different LLMs, we conducted experiments where the watermarker and extractor were instantiated with different LLMs, including **GPT-4o**,

```cpp
vector<int> sumOfAlternates(vector<int> testTuple) {
  // ...
  sum[0] += testTuple[i];
}
```

(a) Original Code

```cpp
vector<int> sumOfAlternates(vector<int> testTuple) {
  // ...
  // Incorrect variable name
  sum[0] += testTuuple[i];
}
```

(b) Watermarked Code

Figure 7: Fail case of $\text{CALS}_{\text{code}}$. "Original" refers to the unwatermarked code.

Method	Metric	MBJP (%)	MBJSP (%)	MBCPP (%)	MBPP (%)
AWT_{code}	BitAcc(%)	93.88	83.97	97.12	/
	Pass(%)	0	0	0	/
$\text{CALS}_{\text{code}}$	BitAcc(%)	93.31	93.50	92.89	/
	Pass(%)	68.65	76.77	68.19	/
SrcMarker	BitAcc(%)	99.44	96.94	96.04	/
	Pass(%)	97.86	97.99	97.64	/
$\text{CodeMark-LLM}_{\text{deepseek}}$	BitAcc(%)	98.69	**99.56**	**98.75**	**98.37**
	Pass(%)	97.25	99.75	98.50	99.75
$\text{CodeMark-LLM}_{\text{gemini}}$	BitAcc(%)	**98.70**	98.19	98.13	97.37
	Pass(%)	**99.40**	**100**	**99.75**	**100**

Table 10: Performance on cross-model generalization. "/" indicates that the method cannot be applied to the Python language.

DeepSeek-V3, and **Gemini 1.5 Pro**. As shown in Table 12, the BitAcc remains consistently above 95% across all combinations, with minimal performance degradation compared to same-model settings. These results demonstrate that our framework maintains stable embedding and extraction accuracy even when different LLMs are used in cross-model scenarios.

C.9 Evaluation of Inference Runtime Cost During Watermark Embedding

To evaluate the inference runtime cost during watermark embedding, we conducted experiments on the CSN-Java dataset. The code lengths were divided into three categories: Short (less than 10 lines), Medium (10-50 lines), and Long (more than 50 lines). Considering that training-based methods consume a significant amount of GPU resources (we used an NVIDIA RTX 4090 GPU for training in our experiments), we adopted a commonly used market rental price ($2/h) as the basis for estimating training costs. To facilitate comparison, we used the **Per-sample Cost** as the evaluation metric. The experimental results, as shown in Table 13, indicate that CodeMark-LLM achieves significantly lower per-sample costs across all code length categories. As the size of the codebase increases, this cost remains lower than that of training-based methods. Moreover, training-based methods have almost no cross-language adaptability because they require specific adapters for each programming language, which significantly reduces the efficiency of traditional methods in large-scale cross-language deployment. In contrast, CodeMark-LLM does not rely on language adapters and naturally adapts to different programming languages, offering higher deployment flexibility. Therefore, CodeMark-LLM demonstrates

```
async function tryApplyUpdates (onHotUpdateSuccess) {
  if (!module.hot) {
    console.error('HotModuleReplacementPlugin is not in Webpack configuration
        .')
    //...
  }
  //...
  function handleApplyUpdates (err, updatedModules) {
    //...
    if (typeof onHotUpdateSuccess === 'function') {
      onHotUpdateSuccess()
    }
    //...
  }
  //...
}
```

(a) Original Code

```
async function tryApplyUpdates ( sizeMap ) {
  if ( ! module.hot ) {
    // Incorrect string literal due to missing spaces
    console.error ( 'HotModuleReplacementPluginisnotinWebpackconfiguration.' ) ;
    //...
  }
  //...
  function handleApplyUpdates ( err , updatedModules ) {
    //...
    // Wrong condition and misplaced function call
    if ( typeof sizeMap = == 'function')  onHotUpdateSuccess ( ) ;
    }
    //...
  }
  //...
}
```

(b) Watermarked Code

Figure 8: Fail case of SrcMarker. "Original" refers to the unwatermarked code.

greater advantages in large-scale cross-language deployment.

C.10 LLM Call Analysis

We measure the average number of LLM calls per sample on four datasets: MBJP, MBPP, MBCPP, and MBJSP. Both the embedding and extraction procedures are evaluated. Table 14 summarizes the results. Across all datasets, the average number of LLM calls remains close to 1.00 for both embedding and extraction. Only a small fraction of samples trigger additional calls due to occasional API-side retries, resulting in a negligible increase in the overall average. These results indicate that CodeMark-LLM requires approximately one LLM inference per sample in practice, demonstrating stable and efficient behavior.

C.11 Robustness Evaluation on MBXP

In this section, we provide supplementary experimental results for the robustness evaluation on MBXP dataset. We use the same setup described in Section 4.5. The results are shown in Table 15, CodeMark-LLM demonstrates strong robustness against both structure-level and identifier-level random removal attacks on the MBXP dataset. It maintains high BitAcc and functionality across Java, JavaScript, and C++ subsets under varying attack intensities. While accuracy slightly degrades as the number of applied transformations or the extent of variable renaming increases, the system remains consistently reliable, even under full variable renaming or multiple code transformations.

Method	2 bit	4 bit	6 bit	8 bit
AWT_{code}	90.50	93.88	85.43	61.67
SrcMarker	99.41	99.44	93.13	81.22
CodeMark-LLM	**99.76**	**99.72**	**97.57**	**95.10**

Table 11: BitAcc changes with respect to watermark capacity (2, 4, 6, and 8 bits) on the MBJP dataset.

Watermarker	Extractor	MBJP (%)	MBJSP (%)	MBCPP (%)	MBPP (%)
GPT-4o	GPT-4o	**99.72**	99.47	**99.64**	**97.85**
	DeepSeek-V3	99.30	**99.72**	98.85	95.51
	Gemini-1.5-pro	99.26	**99.72**	99.48	95.38
DeepSeek-V3	DeepSeek-V3	98.69	**99.56**	**98.75**	98.37
	GPT-4o	95.59	97.62	98.28	96.73
	Gemini-1.5-pro	**99.62**	95.67	98.28	**98.99**
Gemini-1.5-pro	Gemini-1.5-pro	98.70	98.19	98.13	**97.37**
	GPT-4o	98.79	98.43	98.54	96.38
	DeepSeek-V3	**99.34**	**99.75**	**98.98**	96.96

Table 12: Bit Accuracy (BitAcc) of Cross-LLM Watermarking and Extraction Across Datasets

Code Length	Method	Training Time (h)	Embedding Time (s)	Extraction Time (s)	Total Time (h)	Economic Cost ($)
Short	SrcMarker	13.32	**0.0952**	0.0037	13.32	0.0027
	AWT_{code}	61.50	0.1713	**0.0026**	61.50	0.0123
	$CodeMark\text{-}LLM_{deepseek}$	**0**	5.0200	3.6000	27.02	0.0018
	$CodeMark\text{-}LLM_{gemini}$	**0**	1.9000	1.1000	11.15	**0.0011**
	$CodeMark\text{-}LLM_{gpt\text{-}4o}$	**0**	1.4000	1.0000	**5.85**	0.0013
Medium	SrcMarker	13.32	0.0794	0.0032	13.32	0.0027
	AWT_{code}	61.50	**0.0346**	**0.0019**	61.50	0.0123
	$CodeMark\text{-}LLM_{deepseek}$	**0**	8.8000	5.1000	27.02	0.0021
	$CodeMark\text{-}LLM_{gemini}$	**0**	3.1000	1.5000	11.15	**0.0012**
	$CodeMark\text{-}LLM_{gpt\text{-}4o}$	**0**	2.4000	1.2000	**5.85**	0.0017
Long	SrcMarker	13.32	**0.0924**	0.0032	13.32	0.0027
	AWT_{code}	61.50	0.1107	**0.0023**	61.50	0.0123
	$CodeMark\text{-}LLM_{deepseek}$	**0**	24.0000	4.6800	27.02	0.0028
	$CodeMark\text{-}LLM_{gemini}$	**0**	6.6000	1.7000	11.15	**0.0017**
	$CodeMark\text{-}LLM_{gpt\text{-}4o}$	**0**	6.2000	1.8000	**5.85**	0.0029

Table 13: Comparison of Inference Runtime and Economic Cost.

Dataset	Avg. LLM Calls (Embedding)	Avg. LLM Calls (Extraction)
MBJP	1.0083	1.0023
MBPP	1.0010	1.0010
MBCPP	1.0026	1.0026
MBJSP	1.0025	1.0000
Average	**1.0036**	**1.0037**

Table 14: Average number of LLM inference calls per sample for watermark embedding and extraction across different datasets.

Attack	MBJP		MBJSP		MBCPP	
	BitAcc(%)	Pass(%)	BitAcc(%)	Pass(%)	BitAcc	Pass(%)
No Atk.	99.72	99.31	99.47	99.87	99.64	99.35
T@1	88.84	93.92	93.10	98.37	92.64	95.16
T@2	86.84	86.74	88.08	98.12	91.36	92.15
T@3	84.46	82.32	88.05	96.99	89.20	91.23
V@25%	96.03	91.57	95.68	96.49	94.93	90.18
V@50%	95.75	88.81	93.62	93.60	92.67	88.48
V@75%	90.61	87.97	90.90	90.97	91.66	88.09
V@100%	82.29	87.29	87.86	89.34	90.35	86.91

Table 15: Performance under random removal attack