CODEKAST: KEYED AST-SAFE STRUCTURAL WATERMARKING FOR LLM-GENERATED CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

Watermarking code generated by large language models poses the challenge of embedding detectable evidence without compromising functional correctness. Existing methods rely on the assumption that biasing token distributions at generation time is sufficient to leave a persistent watermark signal in code. Our analysis reveals a fundamental limitation of this assumption: program semantics are carried by structure rather than surface tokens, and token-level biases can perturb syntax-critical elements, thereby compromising functional correctness. To address this problem, we propose CodeKAST, a unified framework that embeds key-controlled structural evidence in the abstract syntax tree space rather than the raw token space and uses constrained syntax transformations to generate functional watermarked programs. Specifically, an HMAC-controlled encoding scheme derives a structural state for each supported AST site from a secret key and predeclared verification information; a candidate program is released only after passing syntax checking, round-trip reconstruction, and execution-based screening. During verification, CodeKAST compares the structural states observed in the released code with those expected under a nominated key. The resulting evidence supports both watermark-presence detection and candidate-key attribution, while the verifier abstains when the available sites are insufficient to distinguish candidate keys.On HumanEval+ and MBPP+, CodeKAST achieves the best detection performance among the evaluated methods under a unified model and evaluation protocol, while preserving the functional correctness of the original code, without an extra generative model training process.

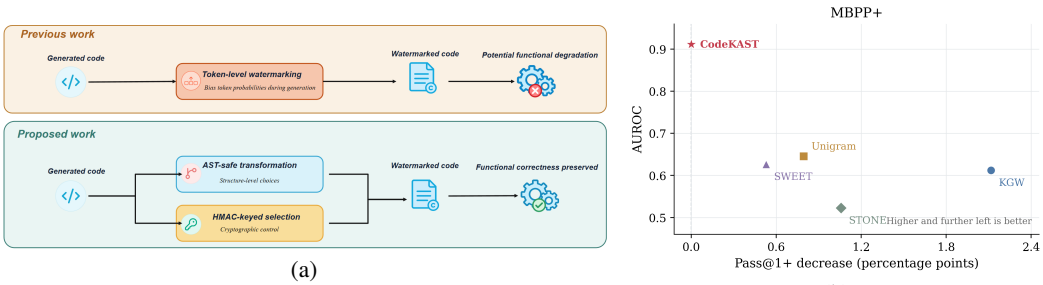
1 INTRODUCTION

With the widespread adoption of large language models for code generation, generated code is increasingly incorporated into practical software-development workflows and code repositories, while the original conversations and server-side logs may not be retained indefinitely. This makes it difficult for model providers to verify code provenance in copyright disputes, service-abuse investigations, or security audits. Digital watermarking provides a practical means of attaching provenance evidence directly to generated code. However, unlike natural language, small modifications to code may alter name resolution, execution order, or return values. A code watermark must therefore provide detectable provenance evidence while preserving functional correctness.

Existing watermarking methods can be broadly grouped into three lines. Token-bias methods adjust decoding probabilities, but do not directly protect the behavior of an individual program (Kirchenbauer et al., 2023; Zhao et al., 2024a). Code-aware generation-time methods use entropy, syntax, or type information, but still require access to generation (Lee et al., 2024; Guan et al., 2024; Kim et al., 2026). AST-assisted approaches identify replaceable components, but may still embed evidence through token choices or rely on trained auxiliary models (Guo & Cheng, 2025; Yu et al., 2025). These constraints motivate a training-free method that can watermark completed code.

To address these challenges, we propose CodeKAST, a training-free post-generation code-watermarking framework that embeds key-controlled structural evidence through constrained AST transformations. As illustrated in Figure 1(a), existing methods primarily embed watermarks by modifying token distributions during generation, which can perturb syntax-critical elements and thereby compromise functional correctness. CodeKAST instead embeds key-controlled structural

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) (b)

Figure 1: Conceptual comparison and main result. (a) Unlike generation-time token watermarking, CodeKAST combines AST-safe transformations with HMAC-keyed structural selection. (b) Detection performance and Pass@1+ decrease on MBPP+; higher and further left is better.

evidence in the abstract syntax tree space rather than the raw token space: constrained AST-safe transformations define structure-level candidate choices, and HMAC-keyed selection determines which state is adopted at each site, producing a watermark that is both detectable and attributable while preserving functional correctness. During verification, CodeKAST recovers the structural choices from the released code and compares them with those expected under a nominated key. The resulting evidence supports watermark-presence detection and, when the program provides sufficient structural capacity, candidate-key attribution. If evidence is absent or multiple keys remain indistinguishable, the verifier abstains rather than forcing an attribution decision. Figure 1(b) further shows that CodeKAST achieves stronger detection performance on MBPP+ while preserving program functionality.

Our main contributions are summarized as follows:

- We propose CodeKAST, a training-free post-generation code-watermarking framework that embeds key-controlled structural evidence through constrained AST transformations and screens candidate programs using syntax and execution checks.
- We design an HMAC-controlled structural encoding scheme and an evidence-consistent verification protocol that separates watermark-presence detection from candidate-key attribution and preserves abstention for ambiguous evidence.
- Experiments on HumanEval+ and MBPP+ show that, under a unified model and evaluation protocol, CodeKAST achieves the best detection performance while preserving the Pass@1 of the original code.

2 Related Work

2.1 Token-Bias Generation-Time Watermarking

Token-based watermarking for LLM outputs controls token selection during generation such that a detector can identify watermarked text through statistical testing (Zhao et al., 2024b). Kirchenbauer et al. (2023) introduced a representative generation-time scheme in which a secret-key-controlled pseudorandom rule partitions the vocabulary and increases the probability of selected tokens during decoding. Subsequent studies explored fixed vocabulary partitions with provable robustness (Zhao et al., 2024a), unbiased or distortion-free alternatives (Hu et al., 2024; Kuditipudi et al., 2024), and computational undetectability under cryptographic assumptions (Christ et al., 2024). WaterMax examined the trade-off among detectability, robustness, and generation quality (Giboulot & Furon, 2024). Later work surveyed these approaches (Lalai et al., 2025) and studied alternative sampling schemes (Mao et al., 2025). Separately, Gu et al. (2025) and He et al. (2025) investigated low-entropy generation and stringent false-positive constraints, where limited token-level freedom reduces usable watermark signal.

Despite these advances, such methods operate primarily on surface-level token distributions. Preserving the distribution of generated text does not imply preservation of the execution behavior of a

particular program. The issue is especially pronounced for low-entropy outputs, where only a small number of plausible next tokens may be available without changing the generated content.

2.2 Code-Aware Generation-Time Watermarking

Code-aware generation-time watermarking exploits code-specific information, including token entropy, grammatical rules, and type information, to identify safer positions for watermark embedding during generation. Lee et al. (2024) proposed SWEET, which excludes low-entropy positions to balance watermark detectability and code quality. Guan et al. (2024) introduced CodeIP, a grammar-guided approach that uses a learned type predictor to encode multi-bit provenance information. Guo & Cheng (2025) proposed ACW, which uses AST analysis to identify replaceable program components and trains a watermark partitioning network to guide key-controlled token grouping during generation. More recently, Kim et al. (2026) introduced STONE, which uses syntax-aware token selection to avoid perturbing syntactically critical elements.

Despite explicitly accounting for code structure, these methods still use tokens as the final embedding carrier. In addition, these generation-time approaches require access to the decoding process, and some rely on learned auxiliary models, making it difficult to attach watermark evidence retrospectively to code that has already been generated. Unlike ACW, which intervenes during generation and relies on a learned partitioning model, CodeKAST operates on completed programs and derives structural choices directly from HMAC without auxiliary training.

2.3 Post-Generation and Structure-Aware Watermarking

Post-generation watermarking attaches evidence to completed outputs. PostMark explores this setting for natural-language text (Chang et al., 2024). RoSeMary combines learned code-watermark embedding and extraction with zero-knowledge verification (Zhang et al., 2025). These approaches differ in their embedding mechanisms and verification objectives; post-generation operation does not itself imply a training-free design. CodeKAST instead uses constrained AST transformations and deterministic keyed choices without training an auxiliary model.

Robustness remains a separate concern: an ICLR Tiny Papers study reports that semantics-preserving code transformations can weaken the code watermarks it evaluated (Suresh et al., 2024). We therefore evaluate detection, candidate-key attribution, and functional correctness separately, rather than treating the embedding stage as a guarantee of robustness.

3 Methodology

3.1 Problem Definition

The goal of code watermarking is to embed detectable evidence into LLM-generated programs while preserving their functional behavior. Existing methods commonly adjust token distributions during generation. To account for the explicit syntactic structure of code, we study a post-generation watermark that can be applied directly to a completed program.

Let x denote a generated program and K a secret key. The verification information $c = (d, t, v)$ contains a deployment identifier, task identifier, and key version, while a domain-separation label s distinguishes different uses of the same key. Given a set of feasible transformations $\mathcal{T}(x)$ and public screening tests P, the encoder constructs a candidate program y. The candidate is released only after passing syntax checking, structural reconstruction, and test execution; otherwise, the original program x is retained.

The verifier uses the released code, a candidate key, and the same verification information to compute an evidence-consistency score $S(y, K, c, s)$. This score supports watermark-presence detection and, given a finite candidate-key set, unique key attribution. If the available evidence is insufficient or multiple keys remain indistinguishable, the verifier abstains.

Threat model. The publisher holds the secret embedding key; an authorized verifier uses candidate keys and the same public verification information to detect and attribute released code. An attacker can inspect and edit released code and knows the watermarking algorithm, but does not

possess the secret key. The attacker's objective is to suppress detection or disrupt attribution while preserving program functionality. Our experiments evaluate formatting and local-variable renaming without adaptive access to a verification oracle. They do not establish robustness against arbitrary semantics-preserving rewrites, key compromise, or adaptive oracle attacks.

3.2 PRELIMINARIES

An abstract syntax tree (AST) represents the syntactic structure of a program and provides the basis for structural transformations. We define an AST location that the encoder can modify and the verifier can recover as a structural site. Each site contains a set of candidate states and is identified by its transformation family and index under an agreed traversal order.

HMAC is a keyed message-authentication mechanism that deterministically maps a secret key and an input message to a hash value (Krawczyk et al., 1997). The same key and input always produce the same result, while the result is difficult to predict without the key. CodeKAST uses HMAC-SHA256 to map the secret key, verification information, and site identity to a target structural state at each AST site.

3.3 HMAC-CONTROLLED STRUCTURAL ENCODING SCHEME

To embed verifiable evidence into a complete program, CodeKAST first parses the generated code and enumerates supported AST sites under a fixed traversal order. Each site is associated with a transformation family f_i and a feasible state set $\mathcal{A}_i$. These states provide different code realizations while preserving the intended functionality. Sites that cannot be transformed safely or recovered reliably by the verifier do not participate in encoding. Figure 2 illustrates this encoding process on a short program.

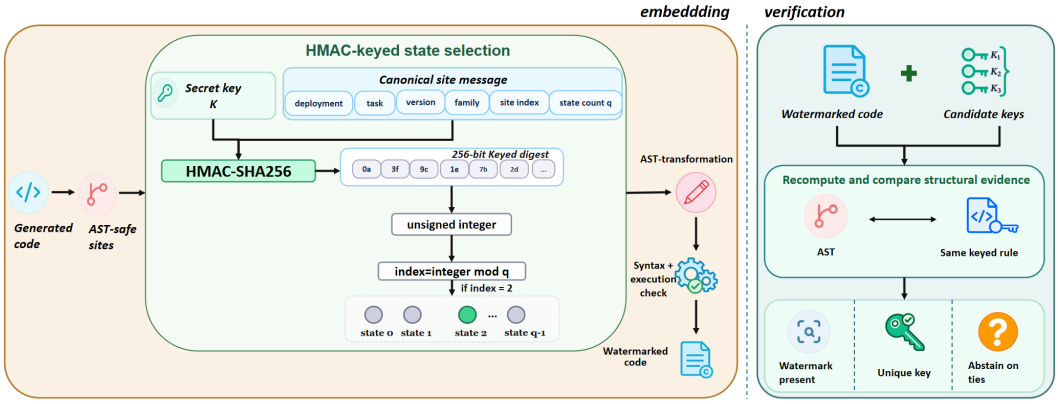


Figure 2: Illustrative HMAC-keyed AST choices for loop and local-variable names. The transformed program is released only after syntax and execution checks.

For site i, let $q_i = |\mathcal{A}_i|$. CodeKAST combines the verification information, domain-separation label, and site identity into

$$m_i = \text{Serialize}(d, t, s, v, f_i, i, q_i), \tag{1}$$

here, Serialize concatenates the fields in the displayed order using the shared canonical format.

and maps the message and secret key to a target structural state:

$$a_i = \text{U64}(\text{HMAC-SHA256}(K, m_i)) \bmod q_i. \tag{2}$$

U64 interprets the first eight digest bytes as a big-endian unsigned 64-bit integer.

The encoder rewrites site i as $\mathcal{A}_i[a_i]$ and proceeds through the remaining sites. Because the encoder and verifier share the same site order, state sets, and message format, the verifier can reconstruct the structural states expected under a candidate key. After rewriting, the candidate program undergoes syntax parsing, round-trip structural reconstruction, and test execution. CodeKAST releases the candidate only when all checks succeed; otherwise, it retains the original program.

Algorithm 1 HMAC-controlled structural encoding

Require: Program x, key K, context (c, s), screening tests P
1: $\mathcal{I} \leftarrow \mathrm{EnumerateSites}(\mathrm{Parse}(x))$
2: $z \leftarrow x$
3: **for** each site $i \in \mathcal{I}$ **do**
4: $\quad a_i \leftarrow \mathrm{HMACChoice}(K, c, s, f_i, i, q_i)$
5: $\quad z \leftarrow \mathrm{ApplyState}(z, i, a_i)$
6: **end for**
7: $v \leftarrow \mathrm{Syntax}(z) \wedge \mathrm{RoundTrip}(z, K, c, s) \wedge \mathrm{Passes}(z, P)$
8: **return** z if v; otherwise x

3.4 Evidence-Consistent Verification Protocol with Abstention

CodeKAST treats watermark existence detection as its primary verification task. Its detection score is not produced by an additionally trained classifier; instead, it measures the consistency between the structural choices observed in the released code and those expected under a nominated key. Given released code y, the verifier follows the encoding rules to recover the actual state at each supported site. This procedure does not require the original program, model activations, or embedding locations recorded during encoding.

For site i, let b_i denote the actual state recovered from the released program y. Given a nominated key K', the verifier recomputes the expected state $a_i(K')$ using Equation 2 and defines the site-level match as

$$h_i(K') = \mathbf{1}\left[b_i = a_i(K')\right]. \tag{3}$$

We use an unweighted score in the main method: each valid structural site contributes one binary match, regardless of its transformation family. Let n be the number of valid sites and $H(K') = \sum_i h_i(K')$ the observed match count. From the state cardinalities, we compute the random-match expectation $E = \sum_i 1/q_i$ and variance $V = \sum_i (1/q_i)(1 - 1/q_i)$. The evidence-consistency score is

$$S(y, K', c, s) = \frac{H(K') - E}{\sqrt{\max(V, \epsilon)}}, \tag{4}$$

where ϵ prevents division by zero. The score measures the standardized deviation of the observed match count from its random-match expectation; a higher score indicates stronger agreement with the nominated key. Existence detection compares this score with a threshold to determine whether the code matches the nominated key's watermark evidence. Candidate-key attribution instead asks which key in a given set best explains the observed structural choices. The verifier computes an evidence-consistency score for each candidate key using the same released code and verification information, then compares these scores. It attributes the code only when structural evidence is present and the highest-scoring key remains unique within the specified tie tolerance; if no sites are available or several keys have indistinguishable scores, it abstains. Algorithm 2 summarizes both decisions.

Algorithm 2 Evidence-consistent verification with abstention

Require: Released code y, nominated key $K_0 \in \mathcal{C}$, candidate set $\mathcal{C}$, context (c, s), threshold τ, tie tolerance ϵ_{tie}

$B \leftarrow \text{RecoverStates}(\text{Parse}(y))$
if $B = \varnothing$ **then return** $(0, \text{abstain})$
for each key $K \in \mathcal{C}$ **do**
    $S_K \leftarrow \text{EvidenceScore}(B, \text{HMACChoices}(K, c, s, B))$
end for
$D \leftarrow \mathbf{1}[S_{K_0} \geq \tau]$
$W \leftarrow \{K \in \mathcal{C} : S_K \geq \max_{J \in \mathcal{C}} S_J - \epsilon_{\text{tie}}\}$
return $(D, \text{sole}(W))$ if $|W| = 1$; otherwise $(D, \text{abstain})$

Let L denote program length, n the number of valid structural sites, and m the number of candidate keys. Verification under one nominated key requires one AST parse and a linear scan of the sites, giving $O(L + n)$ time. Candidate-key attribution reuses the parsed structure and requires $O(L + mn)$ time. Encoding additionally performs round-trip reconstruction and execution-based screening, whose cost is dominated by the selected tests.

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate CodeKAST on four public code-generation benchmark settings: MBPP, MBPP+, HumanEval, and HumanEval+. MBPP contains 974 Python programming problems, while the official MBPP+ release retains 378 tasks and provides expanded tests. HumanEval contains 164 handwritten programming problems, and HumanEval+ augments the same task definitions with a stronger test suite.

Implementation details. Experiments use Qwen2.5-7B-Instruct as the code generator. We compare CodeKAST with KGW, Unigram, SWEET, and STONE using their released implementations or documented settings. Unless otherwise stated, all baseline results are obtained from our local reruns of the publicly available implementations under the same generator, dataset split, and evaluation protocol. MBPP+ and HumanEval+ constitute the main paired comparisons. Experiments run on a server equipped with two NVIDIA RTX 3090 GPUs, while the recorded MBPP+ CodeKAST run uses a tensor-parallel size of one. On this server, excluding LLM generation, CodeKAST processes 378 MBPP+ programs in 14.47 seconds, corresponding to 26.12 programs per second. The mean watermark-injection time is 12.2 ms per program, and the complete post-generation pipeline, including validation and detection, takes 37.4 ms per program on average. The 974-task MBPP cohort is used for development ablations and is reported separately from the official MBPP+ experiment.

Metrics. For watermark existence detection, we report AUROC, Recall@1%FPR, Accuracy@1%FPR, and F1@1%FPR. Functional correctness is measured by Pass@1+ under EvalPlus tests (Liu et al., 2023). For candidate-key attribution, we report coverage, conditional accuracy among non-abstained samples, the number of unique incorrect attributions, and the abstention rate.

Evaluation protocol. For each detection comparison, we select a threshold on the evaluated Base and watermarked scores that maximizes recall subject to an empirical false-positive rate of at most 1%. Thus, Recall@1%FPR is an in-sample operating-point result, not performance measured after calibration on an independent set. The attribution audit ranks the embedding key together with 256 wrong keys generated deterministically from the indexed prefix `wrong-key`; all 257 keys form the candidate set used across programs.

4.2 Comparison Results

ACW relies on a learned auxiliary partitioning network and generation-time intervention. Reproducing it with our shared generator would require separate network training whose data and optimization

Table 1: Existence-detection comparison on MBPP+ and HumanEval+. Baseline results are obtained from local reruns under the shared evaluation protocol.

Method	MBPP+				HumanEval+			
	AUROC	Recall@1%FPR (%)	Accuracy (%)	F1 (%)	AUROC	Recall@1%FPR (%)	Accuracy (%)	F1 (%)
KGW	0.6120	3.44	51.32	6.60	0.6022	4.91	52.15	9.30
Unigram	0.6456	1.85	50.53	3.61	0.6322	4.27	51.83	8.14
SWEET	0.6265	2.65	51.06	5.13	0.6269	4.88	52.13	9.25
STONE	0.5229	1.59	50.40	3.10	0.5110	1.22	50.30	2.40
CodeKAST (Ours)	**0.9118**	**82.54**	**91.01**	**90.17**	**0.9904**	**93.29**	**96.65**	**96.53**

budget are not controlled by the common inference protocol. We therefore discuss ACW in Related Work but exclude it from the head-to-head comparison.

Watermark existence detection. We first compare CodeKAST with four generation-time watermarking methods for existence detection. Table 1 reports the complete metrics, while Figure 1(b) visualizes the relationship between detection performance and functional change on MBPP+. CodeKAST achieves the best AUROC and Recall@1%FPR among the evaluated methods on both benchmarks while preserving the Pass@1+ of the original code.

Strict functional correctness. Table 2 reports strict EvalPlus correctness. Base and CodeKAST obtain identical Pass@1+ on both datasets. This shows that CodeKAST preserves the base model's functional performance on these benchmarks.

Table 2: Strict functional correctness under EvalPlus.

Dataset	Base Pass@1+ (%)	CodeKAST Pass@1+ (%)
MBPP+	69.05	69.05
HumanEval+	64.02	64.02

Candidate-key attribution. Against 256 sampled wrong keys on MBPP+, CodeKAST produces 311 unique correct attributions, no unique wrong attribution, and 67 abstentions. Attribution coverage is 82.28%, with 100% accuracy among decided samples. The abstentions reflect insufficient or tied structural evidence in short programs.

Robustness to code transformations. We evaluate two transformations on the same 378 MBPP+ programs. AST formatting changes 367 watermarked programs but leaves attribution unchanged from clean code. Local-variable renaming changes 245 programs and reduces correct unique attributions from 311 to 293, without an incorrect unique attribution. Per-task strict EvalPlus pass status is unchanged after formatting; the local-variable renaming run preserves the recorded public-test pass status. These results show that CodeKAST retains key-attribution evidence under formatting and is robust to local-variable renaming.

Table 3: Candidate-key attribution under code transformations on 378 MBPP+ programs with 256 sampled wrong keys. Failed attack applications retain the original code and remain in the denominator. Decided accuracy is computed over non-abstained samples.

Transformation	Correct	Wrong	Abstain	Coverage (%)	Dec. Acc. (%)
Clean	311	0	67	82.28	100.00
AST formatting	311	0	67	82.28	100.00
Local-variable renaming	293	0	85	77.51	100.00

4.3 Ablation Study

Using the same 378 MBPP+ base programs, we remove each renaming family from both embedding and verification. Removing a renaming family requires regenerating the watermarked code. Strict

Table 4: End-to-end structural-family ablation on the same 378 MBPP+ base programs.

Configuration	AUROC	Recall@1%FPR (%)	Pass@1+ (%)
Balanced	0.9118	82.54	69.05
Without `local_rename`	0.9132	82.54	69.05
Without `loop_rename`	0.9103	82.28	69.05

Pass@1+ remains 69.05% in all three configurations, with no newly failing tasks under EvalPlus, indicating no observed functional degradation under these tests. AUROC changes little after excluding either `local_rename` or `loop_rename`, and Recall@1%FPR likewise shows almost no decline. All three configurations produce 311 correct unique key attributions and no incorrect ones. Thus, the observed detection and attribution performance does not depend strongly on either renaming family alone.

4.4 Limitations Analysis

Based on the preceding experiments, CodeKAST preserves program functionality and maintains stable detection and attribution under AST formatting and local-variable renaming. Nevertheless, several limitations remain. First, an individual program may contain only a small number of valid structural sites, causing the verifier to abstain when the available evidence is insufficient. Second, although AST formatting leaves the attribution results unchanged, local-variable renaming reduces attribution coverage from 82.28% to 77.51%, while producing no incorrect unique attribution. Finally, our main evaluation focuses on Qwen2.5-7B-Instruct and Python benchmarks, and broader validation across code-generation models and programming languages remains necessary.

5 Conclusion and Future Work

We introduced CodeKAST, a training-free post-generation watermarking framework for generated code. CodeKAST embeds key-dependent structural evidence into completed programs through HMAC-controlled AST-safe transformations and screens candidate programs using syntax checking, round-trip reconstruction, and execution tests. Experiments on HumanEval+ and MBPP+ show that CodeKAST achieves the best detection performance among the evaluated methods while preserving the Pass@1+ of the original code. Its verification protocol additionally supports candidate-key attribution and abstains when the available evidence is insufficient or ambiguous.

Future work will investigate aggregate attribution across programs from the same deployment; accumulating structural evidence before attribution may improve coverage for short programs while retaining abstention in ambiguous cases. We will also study how to reduce the structural footprint left by rewrites, so that watermarked code remains statistically indistinguishable from natural code to an observer without the key, and evaluate CodeKAST against stronger semantics-preserving removal attacks such as redundant-structure elimination and automated refactoring. Finally, we plan to establish independently calibrated low-false-positive-rate thresholds with out-of-sample false-positive guarantees, and to extend CodeKAST to more code-generation models and programming languages such as Java and C++.

AI Use Statement

Generative AI tools were used to assist with code modification, manuscript formatting, the polishing of the text after the initial draft was completed, and the organization of experimental results. All AI-assisted code and manuscript text were reviewed and verified by the authors, who take full responsibility for the final content of this work.

Reproducibility Statement

The algorithms, public verification fields, evaluation metrics, and complexity analysis are specified in Sections 3 and 4. Appendix A documents the experimental configuration, threshold-

selection procedure, functional-correctness evaluation, and candidate-key protocol. Appendix B details structural-site construction and keyed state selection. Appendices C and D provide extended results and clearly separate the official MBPP+ evaluation from historical development analyses. The accompanying anonymous supplementary code contains the embedding, detection, attribution, attack, ablation, and EvalPlus evaluation scripts together with environment and run-configuration files; secret keys and machine-specific paths are excluded.

REFERENCES

Yapei Chang, Kalpesh Krishna, Amir Houmansadr, John Frederick Wieting, and Mohit Iyyer. Postmark: A robust blackbox watermark for large language models. In *Proceedings of EMNLP*, pp. 8969–8987, 2024.

Miranda Christ, Sam Gunn, and Or Zamir. Undetectable watermarks for language models. In *Conference on Learning Theory*, pp. 1125–1139, 2024.

Eva Giboulot and Teddy Furon. Watermax: Breaking the llm watermark detectability-robustness-quality trade-off. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Tianle Gu, Zongqi Wang, Kexin Huang, Yuanqi Yao, Xiangliang Zhang, Yujiu Yang, and Xiuying Chen. Invisible entropy: Towards safe and efficient low-entropy llm watermarking. In *Proceedings of EMNLP*, pp. 6716–6733, 2025.

Batu Guan, Yao Wan, Zhangqian Bi, Zheng Wang, Hongyu Zhang, Pan Zhou, and Lichao Sun. Codeip: A grammar-guided multi-bit watermark for large language models of code. In *Findings of EMNLP*, pp. 9243–9258, 2024.

Zhimeng Guo and Minhao Cheng. Practical and effective code watermarking for large language models. In *Advances in Neural Information Processing Systems*, 2025.

Haiyun He, Yepeng Liu, Ziqiao Wang, Yongyi Mao, and Yuheng Bu. Theoretically grounded framework for llm watermarking: A distribution-adaptive approach. In *Advances in Neural Information Processing Systems*, 2025.

Zhengmian Hu, Lichang Chen, Xidong Wu, Yihan Wu, Hongyang Zhang, and Heng Huang. Unbiased watermark for large language models. In *International Conference on Learning Representations*, 2024.

Jungin Kim, Shinwoo Park, and Yo-Sub Han. Marking code without breaking it: Code watermarking for detecting llm-generated code. In *Findings of EACL*, 2026.

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

Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. Technical Report RFC 2104, RFC Editor, 1997.

Rohit Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. Robust distortion-free watermarks for language models. *Transactions on Machine Learning Research*, 2024.

Harsh Nishant Lalai, Aashish Anantha Ramakrishnan, Raj Sanjay Shah, and Dongwon Lee. From intentions to techniques: A comprehensive taxonomy and challenges in text watermarking for large language models. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 6162–6175. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.343.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. Who wrote this code? watermarking for code generation. In *Proceedings of ACL*, pp. 4890–4911, 2024.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

Minjia Mao, Dongjun Wei, Zeyu Chen, Xiao Fang, and Michael Chau. Watermarking large language models: An unbiased and low-risk method. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7939–7960. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.391.

Tarun Suresh, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Is watermarking llm-generated code robust? In *Tiny Papers at the 12th International Conference on Learning Representations*, 2024.

Zhuohao Yu, Xingru Jiang, Weizheng Gu, Yidong Wang, Qingsong Wen, Shikun Zhang, and Wei Ye. Saemark: Steering personalized multilingual llm watermarks with sparse autoencoders. In *Advances in Neural Information Processing Systems*, 2025.

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

Xuandong Zhao, Prabhanjan Ananth, Lei Li, and Yu-Xiang Wang. Provable robust watermarking for ai-generated text. In *International Conference on Learning Representations*, 2024a.

Xuandong Zhao, Yu-Xiang Wang, and Lei Li. Watermarking for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 5: Tutorial Abstracts)*, pp. 10–11. Association for Computational Linguistics, 2024b. doi: 10.18653/v1/2024.acl-tutorials.6.

A Reproducibility and Evaluation Protocol

This appendix provides the configuration and protocol details needed to interpret and reproduce the main results. It also reports extended MBPP+ results and clearly separates the official 378-task evaluation from earlier analyses on the 974-task MBPP development cohort.

A.1 Main experimental configuration

The official MBPP+ experiment contains 378 paired tasks. CodeKAST uses the balanced set of structural transformation families together with the deterministic encoder–verifier alignment defined in the main paper, a fixed domain-separation label, and key version `v1`. The HMAC secret is supplied through an environment variable and is not stored in released metadata. The run reuses the same Base drafts across CodeKAST and the MBPP+ comparison methods rather than generating a new Base cohort.

The source-generation metadata records temperature 0.2, top-p 0.95, seed 2026, and a 512-token generation limit. The recorded environment uses Python 3.10.20, PyTorch 2.5.1, Transformers 4.48.3, vLLM 0.7.2, and EvalPlus 0.3.1. Experiments run on a server with two NVIDIA RTX 3090 GPUs; the reported MBPP+ CodeKAST run uses a tensor-parallel size of one. These values describe this run and are not asserted for every baseline or historical development artifact.

A.2 Detection and functional-correctness protocol

The main method uses the unweighted evidence score from the paper: every valid site contributes one binary match. For each reported detection comparison, the threshold is selected on the evaluated Base and watermarked scores to maximize recall subject to an empirical false-positive rate of at most 1%. Consequently, Recall@1%FPR is an in-sample operating point rather than a result obtained from an independent calibration set. AUROC is threshold independent.

Functional correctness is evaluated with official EvalPlus execution. Base and watermarked records are paired by task identifier, and evaluation is rejected when task identifiers are missing, duplicated, or misaligned. Table 5 supplements the main correctness table with paired-task counts and the number of watermark-induced failures.

Table 5: Paired strict functional correctness for CodeKAST.

Dataset	Tasks	Base Pass@1+ (%)	CodeKAST Pass@1+ (%)	Added failures
MBPP+	378	69.05	69.05	0
HumanEval+	164	64.02	64.02	0

A.3 Candidate-key protocol

The attribution audit evaluates the embedding key together with 256 deterministic wrong keys. Wrong-key candidate j is derived from the indexed string `wrong-key` and j using SHA-256; the embedding secret is never included in released metadata. All 257 candidates are evaluated for every program. The verifier returns a key only when the highest-scoring candidate is unique within the tie tolerance; otherwise it abstains.

Coverage is the fraction of programs receiving a unique attribution. Conditional accuracy is computed only over non-abstained programs. On clean MBPP+, CodeKAST returns 311 correct unique attributions, zero incorrect unique attributions, and 67 abstentions, corresponding to 82.28% coverage and 100% conditional accuracy.

B Structural-Site Construction and Keyed Selection Details

This appendix complements the mechanism and equations in Section 3.3 with the implementation details needed to reproduce encoder–verifier alignment. It focuses on site eligibility, deterministic identity construction, and the canonical HMAC input rather than repeating the method definition.

Table 6: Structural-site families and their eligibility and recovery rules.

Family	Eligibility condition	State recovered by the verifier
Parameter alias	An aliasable parameter and a collision-free candidate name are available.	The alias binding introduced at function entry.
Loop-variable rename	The loop target has a recoverable binding whose uses can be renamed consistently.	The selected loop-target name and its bound uses.
Local-variable rename	A local binding can be replaced without capture or collision.	The selected local name at its definition and uses.
Return wrapper	A return expression can be routed through a fresh local binding.	The direct-return or wrapper state preceding the return.

B.1 Supported structural sites

CodeKAST applies family-specific eligibility checks before treating an AST location as a structural site. A candidate is excluded when its binding scope cannot be recovered deterministically, when a replacement name conflicts with an existing identifier, or when the corresponding transformation cannot be reconstructed by the verifier. The encoder and verifier use the same candidate sets, transformation preconditions, and family order. Unsupported or ambiguous locations therefore contribute no evidence rather than being assigned an approximate state.

B.2 Deterministic site identity

Site indices are assigned independently within each transformation family. They are determined by the agreed AST traversal and do not depend on source-code line numbers, surface formatting, or a separately stored embedding map. A site identity consists of the family identifier f_i, its family-local ordinal i, and the number of feasible states q_i. The ordinal advances only when a location satisfies the family-specific eligibility checks. The verifier repeats the same traversal on the released program and reconstructs the ordered sequence of recoverable sites. This design explains why formatting changes that preserve the AST leave the reconstructed identities unchanged.

B.3 Canonical HMAC input

The implementation serializes the public verification fields in the fixed order

$$(d, t, s, v, f_i, i, q_i), \tag{5}$$

encodes the resulting message in UTF-8, and prepends the domain-separation string `choice|`. The target state is derived from

$$\text{HMAC-SHA256}(K, \texttt{choice|} \,\|\, m_i). \tag{6}$$

As specified in Section 3.3, the first eight digest bytes are interpreted as a big-endian unsigned integer and reduced modulo q_i. Including the family identifier, family-local ordinal, and state cardinality separates choices made for different site configurations.

B.4 Alignment example

Consider a function containing one eligible loop-variable site and one eligible local-variable site. The two locations receive the identities $(\texttt{loop_rename}, 0, q_1)$ and $(\texttt{local_rename}, 0, q_2)$. Although both have family-local ordinal zero, their family identifiers produce different HMAC messages. After watermarking, the verifier parses the released program, recovers the selected loop and local names, reconstructs the two identities, and compares the observed states with those derived under each candidate key. This process requires neither the original program nor a saved map of embedding locations.

Table 7: Detection and strict functional utility on 378 MBPP+ tasks. Pass@1+ decreases are computed from the unrounded task-level rates.

Method	AUROC	Watermarked Pass@1+ (%)	Pass@1+ decrease (p.p.)
KGW	0.6120	66.93	2.12
Unigram	0.6456	68.25	0.79
SWEET	0.6265	68.52	0.53
STONE	0.5229	67.99	1.06
CodeKAST	**0.9118**	**69.05**	**0.00**

Table 8: Candidate-key attribution under code transformations with 256 wrong-key candidates.

Transformation	Changed	Correct	Wrong	Abstain	Coverage (%)	Cond. acc. (%)
Clean	0	311	0	67	82.28	100.00
AST formatting	367	311	0	67	82.28	100.00
Local-variable renaming	245	293	0	85	77.51	100.00

C Extended Results on MBPP+

C.1 Detection and strict utility

Table 7 combines existence-detection AUROC with paired strict utility for the shared MBPP+ Base cohort. The Pass@1+ decrease is measured in percentage points relative to the common Base Pass@1+ of 69.05%.

C.2 Robustness and attribution

We evaluate AST formatting and scope-safe local-variable renaming on the same 378 watermarked MBPP+ programs. Failed attack applications retain the original code and remain in the denominator. Formatting changes 367 programs and leaves attribution unchanged. Local-variable renaming changes 245 programs and lowers attribution coverage without producing a unique incorrect attribution. Per-task strict EvalPlus status is unchanged under formatting; the local-variable-renaming run preserves the recorded public-test status.

C.3 End-to-end structural-family ablation

For this ablation, each removed family is excluded from both embedding and verification, and the watermarked programs are regenerated from the same 378 Base drafts. Strict Pass@1+ remains 69.05% for every configuration. Each configuration also produces 311 correct unique attributions and no incorrect unique attribution.

D Historical MBPP Development Analyses

The following analyses use historical 974-task MBPP development artifacts rather than the official 378-task MBPP+ cohort. They are score-only analyses on fixed programs and do not regenerate watermarked code. We include them only to document development decisions; they are not used to support the main comparison claims, and their absolute values should not be compared with the official MBPP+ results.

Table 11: Separate score-only family ablation on a 974-task MBPP development artifact. All listed empirical FPR values are 0.00%.

Included score terms	AUROC	Recall@1%FPR (%)
All families	0.7849	60.68
Without `local_rename`	0.8308	69.57
Without `loop_rename`	0.7710	56.44

Table 9: End-to-end structural-family ablation on the paired MBPP+ cohort.

Configuration	AUROC	Recall@1%FPR (%)	Pass@1+ (%)	Correct	Wrong
Balanced	0.9118	82.54	69.05	311	0
Without `local_rename`	0.9132	82.54	69.05	311	0
Without `loop_rename`	0.9103	82.28	69.05	311	0

Table 10: Historical score aggregation on a fixed 974-task MBPP development artifact. Each score uses its own empirical threshold.

Score variant	AUROC	Recall@1%FPR (%)	Actual FPR (%)
Historical family-weighted variant	0.8128	63.04	0.41
Main unweighted score	0.8129	63.04	0.00

E Reproduction Package

The accompanying reproduction package is organized around the following components:

- CodeKAST embedding, detection, attribution, and abstention implementations;
- the exact transformation-family configuration and non-secret public verification fields;
- scripts for existence detection, EvalPlus evaluation, attacks, attribution, and ablations;
- paired task identifiers and run metadata sufficient to detect mixed-run evaluation;
- environment files for the core evaluator and the GPU generation stack; and
- commands for regenerating paper-facing JSON, CSV, and figures without embedding a secret key in command history or metadata.

The private HMAC key, model checkpoints, generated outputs containing restricted artifacts, and machine-specific absolute paths are excluded. Secret values are supplied through environment variables rather than stored in released metadata.