CodeRipple: Wavelet-Based Detection of LLM-Generated Code

Xingyu Yao[1], **Zhendong Mao**[2*], **Quan Wang**[1]
[1]MOE Key Laboratory of Trustworthy Distributed Computing and Service,
Beijing University of Posts and Telecommunications, Beijing, China
[2]University of Science and Technology of China, Hefei, China
{yaoxingyu, wangquan}@bupt.edu.cn, zdmao@ustc.edu.cn

Abstract

Detecting LLM-generated code is crucial for ensuring software provenance, security, reliability, and licensing compliance. Existing training-free detectors, mostly adapted from text-based methods, rely on global statistics of the Token Perplexity Sequence (TPS) and struggle with code. We reveal a key insight: despite the convergence of global statistics, LLM-generated and human-written code differ fundamentally in their local TPS dynamics: the former shows narrow transient spikes while the latter exhibits broad sustained fluctuations. To capture this distinction, we introduce CodeRipple, a novel training-free detection framework that employs wavelet analysis to characterize TPS morphology across scales. It jointly leverages the Stationary Wavelet Transform to model fluctuation shape and the Discrete Wavelet Transform to quantify cross-scale energy distribution. Evaluated on three challenging benchmarks spanning diverse programming languages, multiple generating LLMs, and various evasion strategies, CodeRipple consistently outperforms existing training-free methods, demonstrating its superior effectiveness and generalizability without any model training. Code available at: https://github.com/yaoxingyu77/CodeRipple.

1 Introduction

Large language models (LLMs) have rapidly advanced in their generative capabilities, producing not only fluent natural language but also syntactically correct and executable source code (Du et al., 2024; Gu et al., 2025). As LLM-generated code is increasingly adopted in diverse programming contexts, concerns are growing regarding its provenance, security, reliability, and licensing (Yu et al., 2023; Pan et al., 2024; Liu et al., 2024; Pearce et al., 2025), thereby highlighting the need for effective detection methods (Shi et al., 2025).

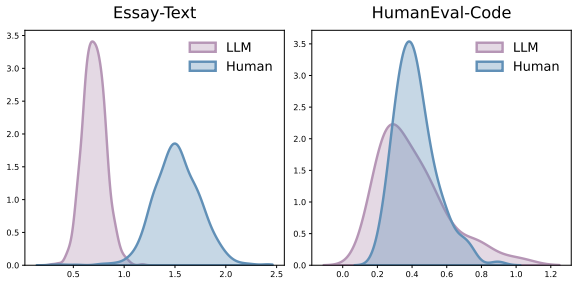

Figure 1: Distribution of the averaged token-wise TPS on Essay (text) (Verma et al., 2024) and HumanEval (code) (Chen et al., 2021), with GPT-3.5-Turbo as the generator and GPT-Neo-2.7B/CodeLlama-7B as proxy models, respectively.

Despite this pressing need, the detection of LLM-generated code remains largely underexplored. Existing approaches fall into two categories: supervised methods, which usually suffer from limited generalization (Bukhari et al., 2023; Idialu et al., 2024; Nguyen et al., 2024), and training-free methods, which are more practical but primarily repurposed from generic LLM-generated text detection techniques rather than being tailored to code (Xu et al., 2025). In practice, current training-free methods typically query a proxy LLM to obtain the **Token Perplexity Sequence** (TPS), defined as the sequence of token-wise negative log-probabilities conditioned on preceding context, which reflects how surprising each token is to the proxy model. Based on this sequence, these methods derive various **global TPS statistics** to distinguish between LLM-generated and human-written code, either by aggregating token-wise TPS values into an overall score (Solaiman et al., 2019; Su et al., 2023), or by quantifying overall TPS sensitivity under random input perturbations (Mitchell et al., 2023).

While effective for natural language, such global TPS statistics lose discriminative power when applied to code (Figure 1). This limitation stems from the constrained nature of code: unlike natu-

*Corresponding author: Zhendong Mao.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 38351–38364

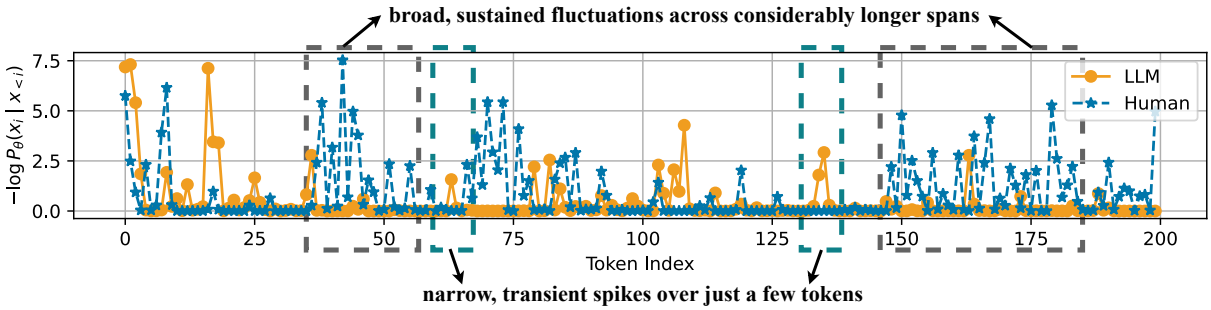

Figure 2: TPS traces for LLM-generated and human-written code solving the same programming task, both sampled from the AI Code Detection Education dataset (Pan et al., 2024).

ral language which exhibits expressive diversity, a programming task admits only a narrow solution space, within which strong syntactic and semantic constraints make LLM-generated and human-written code highly similar, causing their global TPS statistics to converge. Consequently, even code-specific detectors, such as DetectCodeGPT (Shi et al., 2025) which relies on overall TPS sensitivity, struggle in this setting. This exposes a critical gap: global TPS statistics are intrinsically insufficient for distinguishing LLM-generated code in such constrained scenarios.

In this paper, we reveal a fundamental yet previously overlooked phenomenon: despite the convergence of global TPS statistics, LLM-generated and human-written code differ strikingly in their **local TPS dynamics**. As shown in Figure 2, LLM-generated code consistently exhibits narrow, transient TPS spikes over just a few tokens, whereas human-written code displays broad, sustained TPS fluctuations across considerably longer spans. Exploiting this distinction requires a method capable of capturing the morphological structure of TPS fluctuations across neighboring tokens, a capability inherently supported by **wavelet analysis** (Grossmann and Morlet, 1984). In wavelet analysis, the TPS signal is analyzed with wavelets at multiple scales: short-scale wavelets yield strong responses to transient spikes, whereas long-scale wavelets are more sensitive to sustained fluctuations, producing scale-specific signatures that effectively distinguish the two types of code.

Building on this insight, we propose a training-free framework, **CodeRipple**, which detects LLM-generated code by characterizing local TPS fluctuation patterns through two complementary wavelet transforms. First, the **Stationary Wavelet Transform** (SWT) (Nason and Silverman, 1995) captures the morphology of the TPS signal, such as narrow, transient spikes (as in LLM-generated code) and broad, sustained fluctuations (as in human-written code). By preserving translation invariance, SWT enables robust detection of these patterns regardless of their position in the sequence. Second, the **Discrete Wavelet Transform** (DWT) (Mallat, 1989) decomposes the TPS signal into multiple scales and quantifies how response energy is distributed across scales, thereby revealing whether fluctuations are concentrated at fine scales (as in LLM-generated code) or diffuse across coarse scales (as in human-written code). By jointly modeling the two, our approach discriminates LLM-generated code from human-written code based on their distinct TPS signatures: short-span, energy-concentrated versus long-range, energy-diffusive fluctuations.

We evaluate our approach on three challenging datasets: AI Code Detection Education (Pan et al., 2024) that contains ChatGPT-generated code under 13 evasion strategies, e.g., prompt engineering and lexical substitution; HumanEval (Guo et al., 2024) that includes code produced by 5 recently released, high-performance LLMs for code generation, e.g., GPT-4-Turbo and Claude-3-Sonnet; CoDet-M4 (Orel et al., 2025) that comprises LLM-generated code spanning 3 programming languages, namely Python, Java, and C++. Experimental results show that CodeRipple substantially outperforms existing training-free detectors, demonstrating strong effectiveness and generalizability. Our contributions are threefold:

- We identify a fundamental yet previously overlooked distinction in local TPS fluctuation patterns between LLM-generated and human-written code.
- We propose the first wavelet-based, training-free detector that jointly models TPS morphology (via SWT) and cross-scale energy distri-

bution (via DWT) within a unified framework.

- We empirically demonstrate the effectiveness and robustness of our approach across diverse generating LLMs, evasion strategies, and programming languages.

2 Related Work

We review related work on LLM-generated text detection, code detection, and wavelet analysis.

LLM-Generated Text Detection LLM-generated text detection methods generally fall into training-based and training-free paradigms. Training-based methods typically learn discriminative classifiers (Bhattacharjee et al., 2023; Li et al., 2023) but often suffer from poor out-of-distribution generalization (Chakraborty et al., 2024). This has motivated training-free methods that exploit intrinsic statistical discrepancies, evolving from simple global metrics such as log-likelihood (Solaiman et al., 2019) to distributional discrepancy frameworks like DetectGPT (Mitchell et al., 2023) and more efficient variants (Bao et al., 2024). Binoculars (Hans et al., 2024) further improves performance by contrasting perplexity scores from two proxy models. However, these methods are tailored to natural language text and do not directly extend to LLM-generated code, whose rigid syntax and strong structural regularities fundamentally alter token probability statistics.

LLM-Generated Code Detection Existing approaches for LLM-generated code detection also include both supervised and training-free methods. Supervised methods typically rely on handcrafted code features or fine-tuned pretrained code models and require labeled training data, which limits their generalization across models and domains (Bukhari et al., 2023; Idialu et al., 2024; Nguyen et al., 2024). Training-free methods typically operate by analyzing token-level probability statistics generated by proxy models. DetectCodeGPT (Shi et al., 2025) is a representative approach, which detects LLM-generated code by measuring the sensitivity of the Token Perplexity Sequence (TPS) under syntactic perturbations such as inserting spaces and newlines, without requiring external LLM-based perturbations. However, despite these adaptations in the perturbation strategy, DetectCodeGPT still relies on overall TPS sensitivity and struggles to reliably distinguish LLM-generated code in constrained settings (Xu et al., 2025).

Wavelet Analysis Wavelet analysis decomposes a signal into multi-scale components via localized basis functions, ensuring the examination of both transient spikes and persistent trends (Grossmann and Morlet, 1984). It has been successfully applied to various time-series tasks, such as anomaly detection, physiological signal analysis, and financial modeling, where both transient spikes and persistent trends carry meaningful information (Yu et al., 2025; Chen et al., 2025; Masserano et al., 2024). However, to our knowledge, no prior work has applied wavelet analysis to token perplexity sequence for LLM-generated code detection. Our method is the first to demonstrate that multi-scale wavelet features reveal fundamental and highly discriminative patterns in LLM-generated code.

3 Methodology

CodeRipple is a training-free detector that distinguishes LLM-generated code from human-written code by modeling local TPS dynamics with wavelet transforms. As shown in Figure 3, it comprises four stages: 1) constructing the raw TPS signal; 2) modeling TPS morphology with the Stationary Wavelet Transform; 3) modeling TPS energy distribution with the Discrete Wavelet Transform; 4) fusing the resulting features for final detection.

3.1 Constructing Token Perplexity Sequence

Given a code snippet to be detected, we feed it into a proxy LLM with parameters θ and obtain its token sequence $\mathbf{x} = (x_1, \ldots, x_N)$. The proxy model assigns to each token x_i a conditional probability $P_\theta(x_i \mid x_{<i})$, where $x_{<i} = (x_1, \ldots, x_{i-1})$ denotes the preceding context (with $x_{<1}$ interpreted as the initial state). The *Token Perplexity Sequence* (TPS) is defined as the sequence of token-wise negative log-probabilities:

$$s_i = -\log P_\theta(x_i \mid x_{<i}), \tag{1}$$

where s_i measures how surprising x_i is to the proxy model given its preceding context. We denote the full TPS as $\mathbf{s} = (s_1, \ldots, s_N)$.

3.2 Modeling TPS Morphology with SWT

This module applies the *Stationary Wavelet Transform* (SWT) to characterize multi-scale fluctuation morphology in the TPS signal. SWT analyzes the sequence using a fixed-shape filter that expands to broader contexts at coarser scales while preserving the sequence length, enabling consistent comparison of morphological features across scales.

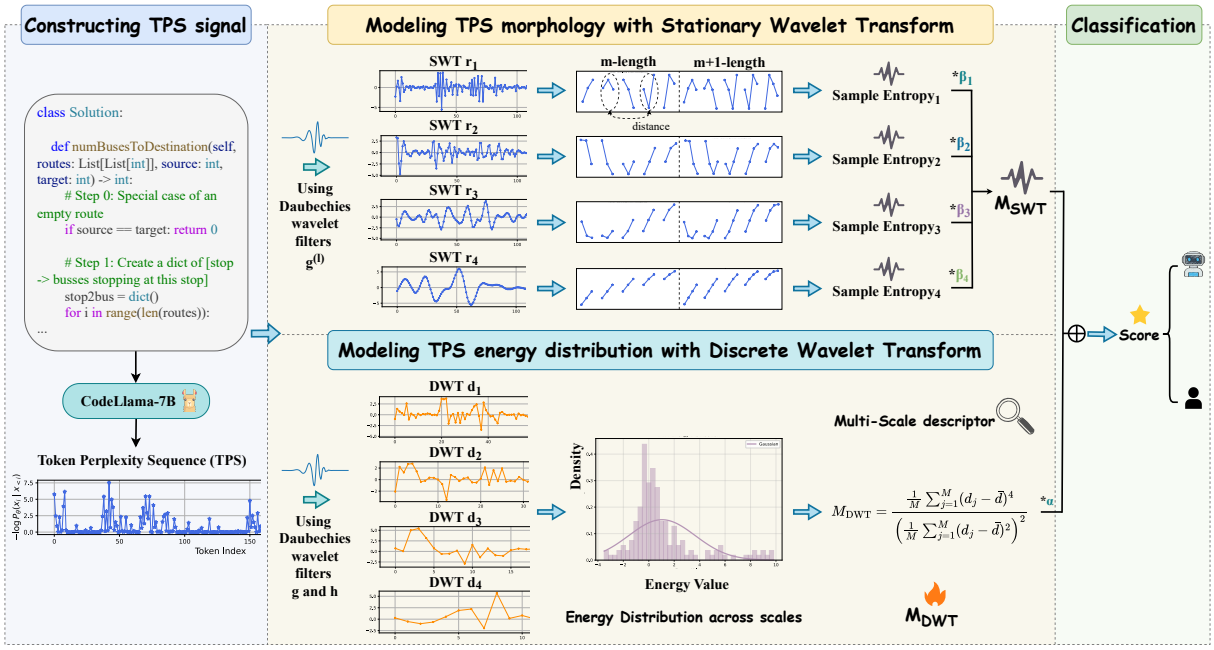

Figure 3: Overall architecture of CodeRipple. It first constructs the raw TPS signal, then models TPS morphology and energy distribution using SWT and DWT, respectively, and finally fuses the resulting features for detection.

Specifically, we consider a Daubechies wavelet (Daubechies, 1988) with high-pass wavelet filter $\mathbf{g}$ and low-pass scaling filter $\mathbf{h}$, both of length p (see Appendix A for details). We use $\mathbf{g}$ as the base filter to analyze the TPS signal at L scales. For each scale $\ell \in \{1, \ldots, L\}$, we construct a dilated filter $\mathbf{g}^{(\ell)}$ of length $K_\ell = 2^{\ell-1}(p-1)+1$ by inserting $2^{\ell-1}-1$ zeros between consecutive entries of $\mathbf{g}$. Formally, for $k = 1, \ldots, K_\ell$, we have:

$$g_k^{(\ell)} = \begin{cases} g_{1+(k-1)/2^{\ell-1}}, & \text{if } 2^{\ell-1} \mid (k-1), \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

Note that $\mathbf{g}^{(1)} = \mathbf{g}$ and $K_1 = p$. We then convolve the TPS signal $\mathbf{s} = (s_1, \ldots, s_N)$ with $\mathbf{g}^{(\ell)}$ (using zero-padding at boundaries) to compute a response that measures local fluctuation strength at scale ℓ:

$$r_{\ell,n} = \sum_{k=1}^{K_\ell} s_{n+k-1} \cdot g_k^{(\ell)}, \quad n = 1, \ldots, N. \quad (3)$$

We denote the response as $\mathbf{r}_\ell = (r_{\ell,1}, \ldots, r_{\ell,N})$, where larger ℓ corresponds to analysis over broader token spans.

For each scale ℓ, we quantify the morphological complexity of $\mathbf{r}_\ell$ using sample entropy (Richman and Moorman, 2000), a standard measure of time-series irregularity. This involves extracting all overlapping subsequences of length m from $\mathbf{r}_\ell$ and assessing their similarity via Chebyshev distance: two subsequences are deemed similar if their distance is at most ϵ. Sample entropy is then defined as the negative logarithm of the conditional probability that two similar subsequences remain similar when extended to length $m+1$ (see Appendix B for details). Higher values indicate more irregular fluctuations and thus greater morphological complexity at scale ℓ. An overall multi-scale complexity measure is obtained by aggregating across scales:

$$M_{\text{SWT}} = \sum_{\ell=1}^{L} \beta_\ell \, \text{SE}_\ell(m, \epsilon), \quad (4)$$

where $\text{SE}_\ell(m, \epsilon)$ denotes the sample entropy of $\mathbf{r}_\ell$ computed with dimension m and tolerance threshold ϵ, and β_ℓ is a hyperparameter weight assigned to scale ℓ.

3.3 Modeling TPS Energy Diffusion with DWT

This module further applies the *Discrete Wavelet Transform* (DWT) to characterize cross-scale fluctuation energy diffusion in the TPS signal. Unlike SWT, DWT analyzes the sequence using fixed filters and progressively downsamples it at coarser scales, enabling energy-preserving decomposition and exact scale-wise energy quantification.

Specifically, we employ the same Daubechies wavelet, defined by high-pass wavelet filter $\mathbf{g}$ and low-pass scaling filter $\mathbf{h}$ of length p. Starting from the original TPS signal $\mathbf{s}^{(0)} = \mathbf{s}$, we perform an

L-level DWT by recursively filtering with $\mathbf{h}$ and $\mathbf{g}$ followed by dyadic downsampling, yielding approximation sequences $\{\mathbf{s}^{(\ell)}\}_{\ell=0}^{L}$ and detail coefficients $\{\mathbf{d}_\ell\}_{\ell=1}^{L}$, where each $\mathbf{s}^{(\ell)}$ and $\mathbf{d}_\ell$ has length $N_\ell = \lfloor N_{\ell-1}/2 \rfloor$ with $N_0 = N$. Formally, for $\ell \in \{1, \ldots, L\}$, the coefficients are computed as:

$$s_n^{(\ell)} = \sum_{k=1}^{p} s_{2n+k-2}^{(\ell-1)} \cdot h_k, \quad n = 1, \ldots, N_\ell, \quad (5)$$

$$d_{\ell,n} = \sum_{k=1}^{p} s_{2n+k-2}^{(\ell-1)} \cdot g_k, \quad n = 1, \ldots, N_\ell, \quad (6)$$

with zero-padding for out-of-range indices. Larger ℓ captures fluctuations over broader token spans.

For each scale ℓ, we retain the signed detail coefficients $\{d_{\ell,n}\}$ to preserve both magnitude and direction of local TPS fluctuations. Aggregating all coefficients across scales and positions yields a single sequence $\mathcal{D} = \{d_j\}$. We summarize its distributional shape using kurtosis, a standard measure of tail concentration, as the multi-scale descriptor:

$$M_{\mathrm{DWT}} = \frac{\frac{1}{M}\sum_{j=1}^{M}(d_j - \bar{d})^4}{\left(\frac{1}{M}\sum_{j=1}^{M}(d_j - \bar{d})^2\right)^2}, \quad (7)$$

where $\bar{d}$ is the mean of $\mathcal{D}$ and $M = |\mathcal{D}|$. High values indicate that fluctuation energy is concentrated in narrow spikes, while low values reflect a more even energy distribution, providing a compact characterization of cross-scale energy diffusion in the TPS signal.

3.4 Fusing Features for Classification

To discriminate whether the input code snippet x is LLM-generated or human-written, we fuse two complementary characterizations of its TPS: the morphology of multi-scale fluctuations (captured by M_{SWT}) and the cross-scale diffusion of fluctuation energy (captured by M_{DWT}). The fused score is defined as:

$$\mathrm{Score} = M_{\mathrm{SWT}} + \alpha\, M_{\mathrm{DWT}}, \quad (8)$$

where α is a hyperparameter balancing the two components. We classify x as LLM-generated if $\mathrm{Score} \geq \tau$, and as human-written otherwise.

4 Experiments

This section evaluates CodeRipple for training-free detection of LLM-generated code against previous state-of-the-arts, and includes extensive analyses to further understand its behavior.

4.1 Experimental Setups

Datasets We conduct evaluations on three challenging and widely-adopted datasets, encompassing human-written and LLM-generated code that spans various programming languages, generating LLMs, and evasion strategies.

- AI Code Detection Education (Pan et al., 2024)[1] contains 5,069 human-written Python problem-solution pairs (problems from Kaggle, solutions from Quescol) and 65,897 LLM-generated Python samples produced by ChatGPT in response to the same Kaggle problems, using 13 evasion strategies. These strategies simulate realistic attempts to evade detection, for example by prompting ChatGPT to mimic human coding style or renaming variables, thus resulting in diverse and challenging LLM-generated samples.

- HumanEval (Guo et al., 2024)[2] comprises 164 Python tasks, each defined by a function signature and a natural language description. This restricted solution space makes it particularly challenging to differentiate human and LLM code. The human-written reference implementations are treated as authentic human samples (Chen et al., 2021). LLM-generated counterparts are obtained by prompting five LLMs, i.e., GPT-3.5-Turbo, GPT-4-Turbo, Claude-3-Sonnet, Claude-3-Opus, and Gemini-1.0-Pro, with the given function descriptions, yielding one sample per task for each model.

- CoDet-M4 (Orel et al., 2025)[3] is a multilingual dataset comprising human-written and LLM-generated code spanning Python, Java, and C++. The samples are sourced from diverse platforms (e.g., LeetCode, CodeForces, GitHub) and are produced by various state-of-the-art LLMs. The test suite, which consists of 17,214 Python, 16,517 Java, and 13,305 C++ samples, is employed for evaluation.

Baselines We compare CodeRipple with six prior training-free detectors. The first five, i.e., LogP (Solaiman et al., 2019), GLTR (Gehrmann et al., 2019),

[1] https://figshare.com/articles/dataset/Replication_Package/24298036

[2] https://github.com/MarkGHX/BiScope/tree/main/Dataset/Code

[3] https://huggingface.co/datasets/DaniilOr/CoDET-M4

Table 1: AUROC of different methods on AI Code Detection Education. The **best** and second-best performances for each evasion strategy are bolded and underlined, respectively.

Detectors	Evasion Strategies													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**AVG**
LogP	0.670	0.729	0.739	0.683	0.761	0.774	0.879	0.535	0.579	0.520	0.716	0.652	0.608	0.680
GLTR	0.647	0.721	0.733	0.661	0.721	0.764	0.862	0.531	0.551	0.523	0.682	0.619	0.575	0.661
LRR	0.517	0.587	0.600	0.505	0.567	0.622	0.642	0.502	0.613	0.515	0.541	0.569	0.593	0.567
FastDetectGPT	0.800	0.825	0.820	0.801	0.843	0.900	0.931	0.611	0.674	0.708	0.834	0.805	0.511	0.774
Binoculars	0.752	0.828	0.821	0.759	**0.861**	**0.907**	**0.934**	0.553	0.568	0.624	0.769	0.734	0.511	0.740
DetectCodeGPT	0.600	0.657	0.622	0.608	0.568	0.591	0.625	0.519	0.575	0.527	0.599	0.575	0.654	0.594
PHD	0.503	0.529	0.529	0.502	0.618	0.552	0.701	0.576	0.503	0.566	0.534	0.522	0.633	0.559
CodeLength	0.765	0.712	0.717	0.762	0.629	0.669	0.504	0.832	0.790	0.845	0.741	0.781	**0.659**	0.723
CodeRipple	**0.925**	**0.871**	**0.881**	**0.922**	0.786	0.874	0.809	**0.860**	**0.871**	**0.908**	**0.937**	**0.943**	0.568	**0.858**

Table 2: AUROC of different methods on HumanEval and CoDet-M4. The **best** and second-best performances are bolded and underlined, respectively. LLM abbreviations: G-4.0 (GPT-4.0-Turbo), G-3.5 (GPT-3.5-Turbo), C-Opus (Claude-3-Opus), C-Sonnet (Claude-3-Sonnet), and Gemi-Pro (Gemini-1.0-Pro).

Detectors	HumanEval						CoDet-M4		
	G-4.0	**G-3.5**	**C-Opus**	**C-Sonnet**	**Gemi-Pro**	**AVG**	**Python**	**C++**	**Java**
LogP	0.577	0.611	0.618	0.585	0.524	0.583	0.544	0.732	0.711
GLTR	0.545	0.576	0.597	0.558	0.517	0.559	0.539	0.734	0.725
LRR	0.624	0.625	0.556	0.570	0.526	0.580	0.513	0.689	0.728
FastDetectGPT	0.579	0.600	0.594	0.585	0.502	0.572	0.645	0.821	0.777
Binoculars	0.567	0.592	0.582	0.564	0.531	0.567	0.603	0.823	0.806
DetectCodeGPT	0.667	0.660	0.708	0.676	**0.540**	0.650	0.556	0.631	0.558
PHD	0.524	0.561	0.569	0.541	0.519	0.543	0.515	0.550	0.562
CodeLength	0.829	0.870	0.702	0.731	0.502	0.727	0.736	0.632	0.558
CodeRipple	**0.865**	**0.901**	**0.759**	**0.772**	0.521	**0.764**	**0.828**	**0.835**	**0.812**

LRR (Su et al., 2023), FastDetectGPT (Bao et al., 2024), and Binoculars (Hans et al., 2024), are general-purpose methods originally designed for LLM-generated text, while DetectCodeGPT (Shi et al., 2025) is specifically tailored for code. These methods are primarily based on global TPS statistics. We further take Persistence Homology Dimension (PHD) (Tulchinskii et al., 2023) as a geometry-based baseline. While the original work employs a supervised classifier on PHD, we directly apply thresholding to the raw PHD scores to match the training-free setting. We also consider code length (number of characters) as a simple heuristic baseline. Overall, all baselines rely on global statistics or structures, whereas CodeRipple leverages local TPS dynamics.

Implementations For a fair comparison, we use CodeLlama-7B (Rozière et al., 2024) as the proxy LLM to extract TPS where applicable. The maximum sequence length is fixed at 256 for AI Code Detection Education and HumanEval, and 512 for CoDet-M4. In CodeRipple, we use Daubechies-6 (db6) wavelets with $p = 12$, and set the decomposition level to $L = 4$ for both SWT and DWT. Sample entropy is computed with subsequence length $m = 3$ and tolerance $\epsilon = 0.2 \times \sigma_{\mathbf{r}_\ell}$, where $\sigma_{\mathbf{r}_\ell}$ is the standard deviation of sequence $\mathbf{r}_\ell$. The overall score combines DWT and scale-dependent SWT components, with weights subject to the constraint $\alpha + \sum_{\ell=1}^{4} \beta_\ell = 1$. Unless otherwise specified, we set $\alpha = 0.01$ with the remaining weights assigned such that $\beta_1 : \beta_2 : \beta_3 : \beta_4 = 4 : 3 : 2 : 1$. For the baselines, FastDetectGPT employs CodeLlama-7B as its sampling model, while Binoculars utilizes CodeLlama-7B-Instruct as the performer. All other configurations follow their respective official implementations. Inference is conducted on NVIDIA A40 GPUs, and as all methods are training-free, results are reported from a single run.

4.2 Main Results

Tables 1 and 2 present the AUROC scores for various methods across the AI Code Detection Education, HumanEval, and CoDet-M4 benchmarks. We use AUROC as the evaluation metric because (1) the former two benchmarks do not provide a validation set for threshold selection, and (2) AUROC offers a comprehensive assessment of model performance across all possible decision thresholds.

Table 3: Ablation results (AUROC) on AI Code Detection Education dataset.

Variants	Evasion Strategies													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**AVG**
CodeRipple	0.925	0.871	0.881	0.922	0.786	0.874	0.809	0.860	0.871	0.908	0.937	0.943	0.568	0.858
w/o SWT	0.881	0.815	0.834	0.882	0.767	0.818	0.715	0.846	0.862	0.884	0.895	0.905	0.663	0.828
w/o DWT	0.913	0.854	0.863	0.911	0.748	0.871	0.828	0.828	0.837	0.881	0.928	0.924	0.507	0.838
replace SWT	0.784	0.725	0.726	0.779	0.629	0.669	0.501	0.796	0.800	0.800	0.766	0.798	0.683	0.727
replace DWT	0.841	0.814	0.814	0.840	0.733	0.729	0.671	0.812	0.839	0.806	0.832	0.847	0.560	0.780
scale-fused SWT	0.922	0.867	0.880	0.922	0.761	0.873	0.818	0.868	0.876	0.910	0.936	0.941	0.562	0.857
scale-split DWT	0.921	0.867	0.873	0.916	0.764	0.865	0.814	0.848	0.860	0.897	0.930	0.935	0.530	0.848

Results across Evasion Strategies Table 1 lists performance under 13 evasion strategies. On Strategy 1 (unmodified LLM-generated code), CodeRipple achieves an AUROC of 0.925, surpassing the second-best method, FastDetectGPT (0.800), by an absolute margin of 0.125. Among the remaining 12 strategies, it ranks first in 8 cases. Averaged across all strategies, it attains a mean AUROC of 0.858, surpassing all competitive baselines and confirming its robustness to diverse evasion techniques.

A notable exception occurs in Strategies 5–7, where CodeRipple underperforms relative to Binoculars. These strategies are highly specialized, requiring ChatGPT to generate assertions, test cases, and unit tests, respectively. Such rigid structural constraints suppress the natural fluctuations in TPS signals, thereby diminishing the discriminative power of our wavelet-based features which depend on subtle morphological and energy variations. In contrast, Binoculars, which leverages cross-model perplexity discrepancies, remains less sensitive to these syntactic regularities, maintaining an advantage in such template-dominated scenarios.

Results across Code Generating LLMs Table 2 reports the performance across code generated by five LLMs. CodeRipple consistently outperforms all baselines on four models, achieving a mean AUROC of 0.764 and demonstrating robust cross-model generalization. Notably, the simplistic CodeLength baseline remains surprisingly competitive on this dataset, yielding a mean AUROC of 0.727.

Results across Programming Languages Table 2 also presents the performance across three programming languages: Python, C++, and Java.[4] CodeRipple consistently achieves the top performance in all three cases, highlighting its robust cross-language generalization. However, CodeRipple yields more pronounced gains in Python than in C++ or Java. This disparity likely stems from Python's greater syntactic flexibility, which facilitates richer TPS fluctuation patterns. Conversely, the structural rigidity of C++/Java imposes stricter constraints on token generation, inherently limiting the signal variance captured by wavelet analysis.

[4]Due to divergent dataset characteristics, the combination weights (α and β's) used for AI Code Detection Education and HumanEval are not applicable to CoDet-M4. We therefore perform hyperparameter tuning on the CoDet-M4 development set and report the final performance on its test set.

4.3 Ablation Studies

We perform ablation studies on CodeRipple's key components, considering the following variants:

- w/o SWT: Removes the SWT branch and uses only DWT branch M_{DWT} for detection.
- w/o DWT: Removes the DWT branch and uses only SWT branch M_{SWT} for detection.
- replace SWT: Replaces the SWT branch with direct sample entropy computation on raw TPS and uses it alone for detection.
- replace DWT: Replaces the DWT branch with direct kurtosis computation on raw TPS and uses it alone for detection.
- scale-fused SWT: Fuses the four SWT scales $\{\mathbf{r}_\ell\}_{\ell=1}^4$ into a single response and computes sample entropy on the merged signal.
- scale-split DWT: Computes kurtosis per DWT scale $\{\mathbf{d}_\ell\}_{\ell=1}^4$ and merges the results into a single score using a weighting of $4:3:2:1$.

For scale-fused SWT and scale-split DWT, the morphology and energy scores are fused with a fixed weighting of $0.99{:}0.01$ for final detection.

Table 3 presents ablation results on the AI Code Detection Education dataset. We observe that: (1)

Table 4: AUROC results of CodeRipple under different wavelet functions on AI Code Detection Education dataset.

Wavelets	Evasion Strategies													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**AVG**
db4	0.921	0.875	0.882	0.920	0.789	0.869	0.799	0.852	0.847	0.889	0.921	0.930	0.569	0.851
db6	0.925	0.871	0.881	0.922	0.786	0.874	0.809	0.860	0.871	0.908	0.937	0.943	0.568	0.858
db8	0.925	0.872	0.884	0.924	0.785	0.883	0.831	0.859	0.871	0.899	0.940	0.942	0.569	0.860
db10	0.923	0.871	0.876	0.922	0.777	0.876	0.827	0.855	0.861	0.899	0.934	0.938	0.568	0.856
sym4	0.906	0.867	0.868	0.904	0.757	0.846	0.780	0.857	0.867	0.893	0.914	0.919	0.551	0.841
sym6	0.914	0.874	0.881	0.914	0.778	0.867	0.802	0.853	0.860	0.894	0.921	0.928	0.557	0.849
sym8	0.920	0.869	0.883	0.917	0.784	0.873	0.833	0.856	0.860	0.903	0.932	0.934	0.579	0.857
sym10	0.921	0.867	0.879	0.920	0.789	0.875	0.828	0.859	0.861	0.904	0.930	0.934	0.584	0.858

Table 5: AUROC results of CodeRipple under different proxy models on AI Code Detection Education dataset.

Proxy LLMs	Evasion Strategies													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**AVG**
Llama3.2-3B	0.875	0.842	0.851	0.871	0.676	0.815	0.728	0.819	0.810	0.829	0.880	0.885	0.511	0.799
Llama2-7B	0.912	0.860	0.868	0.910	0.791	0.861	0.785	0.872	0.868	0.910	0.914	0.923	0.620	0.853
CodeLlama-7B	0.925	0.871	0.881	0.922	0.786	0.874	0.809	0.860	0.871	0.908	0.937	0.943	0.568	0.858

Compared to the full model CodeRipple, the w/o SWT and w/o DWT variants consistently degrade performance, with absolute mean AUROC drops of 0.030 and 0.020, respectively. This indicates that both branches provide complementary signals, and their combination yields optimal performance. (2) Compared to the w/o SWT variant (which relies solely on DWT), the replace DWT variant performs substantially and consistently worse, yielding an absolute mean AUROC drop of 0.048. This verifies the necessity of DWT: without this transform, computing kurtosis on raw TPS alone is insufficient for effective detection. (3) Compared to the w/o DWT variant (which relies solely on SWT), the replace SWT variant performs substantially and (almost) consistently worse, resulting in an absolute mean AUROC drop of 0.111. This verifies the necessity of SWT: without this transform, using sample entropy on raw TPS alone is insufficient for effective detection. (4) Compared to the full model, scale-fused SWT and scale-split DWT perform slightly worse. This suggests that, for modeling morphology, computing features separately at each scale is more effective, whereas for capturing energy diffusion, aggregating information across all scales yields better results.

4.4 Additional Analyses

Effect of Wavelet Function We investigate the robustness of CodeRipple to the choice of wavelet function, evaluating members of the Daubechies (db) and Symlet (sym) families across different orders, where the order directly determines the filter length. As shown in Table 4, CodeRipple achieves consistently strong detection performance across all tested wavelets, with AUROC values confined to a rather narrow range. This indicates that CodeRipple is not sensitive to the choice of wavelet function. The performance gains primarily stem from the proposed modeling of local TPS dynamics, rather than from careful selection of the wavelet.

Effect of Proxy Model We further investigate the robustness of CodeRipple to the choice of proxy LLM used to construct raw TPS signal. We evaluate two general-purpose models, Llama3.2-3B and Llama2-7B, along with a code-specialized model, CodeLlama-7B. As shown in Table 5, both 7B models consistently and substantially outperform the 3B model, indicating that larger proxy models generally lead to improved detection performance. Between the two 7B models, CodeLlama-7B achieves marginally higher AUROC than Llama2-7B, suggesting a modest benefit from code specialization. Overall, the minor performance gap between the two 7B variants demonstrates that CodeRipple is not sensitive to the specific choice of proxy model, provided that the model is sufficiently large.

Effect of Weighting Strategy We examine the effect of different weighting strategies for combining SWT and DWT components. First, we fix the DWT weight to $\alpha = 0.01$ and evaluate 9 representative strategies for aggregating multi-scale SWT features, each emphasizing different scales to vary-

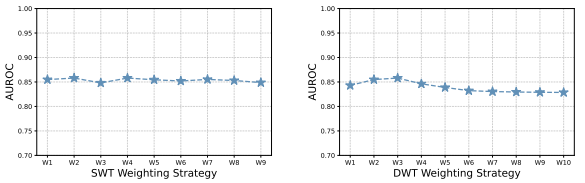

Figure 4: Average AUROC of CodeRipple across 13 evasion strategies with different SWT (left) and DWT (right) weighting strategies on AI Code Detection Education. Detailed results in Tables 7 and 8 (Appendix C).

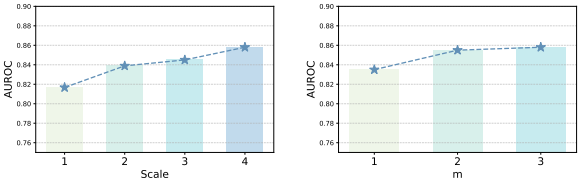

Figure 5: Average AUROC of CodeRipple across 13 evasion strategies on AI Code Detection Education dataset, using different numbers of wavelet scales L (left) and different subsequence lengths m (right). Detailed results in Tables 9 and 10 (Appendix D).

ing degrees (details in Appendix C). Figure 4 (left) shows that CodeRipple achieves consistently high average AUROC across all 9 weighting strategies, demonstrating its robustness to the choice of SWT scale weights. Then, we vary the DWT weight α across 10 settings ranging from 0.001 to 0.850 (details in Appendix C), and fix the relative weights of the SWT scales to $\beta_1 : \beta_2 : \beta_3 : \beta_4 = 4 : 3 : 2 : 1$ with $\alpha + \sum_{\ell=1}^{4} \beta_\ell = 1$. Figure 4 (right) shows that the average AUROC remains rather stable across all 10 weighting strategies, demonstrating CodeRipple's robustness to the choice of DWT weight.

Effect of Other Hyperparameters Figure 5 (left) shows CodeRipple's performance with $L \in \{1, 2, 3, 4\}$ wavelet scales. The average AUROC consistently improves as L increases, confirming the benefit of multi-scale TPS modeling. However, more scales also add complexity and uncertainty in scale combination. We thus choose $L = 4$ for a balance between performance and stability. Figure 5 (right) further shows that CodeRipple's performance is insensitive to the subsequence length m for sample entropy computation.

Inference Costs Table 6 compares CodeRipple with the baselines in terms of average inference time (ms/sample) and peak VRAM usage. Despite its strong performance gains, CodeRipple incurs only modest inference overhead. It is 2–2.5×slower than the simple baselines LogP, GLTR, and LRR with nearly identical VRAM; it almost matches the speed of dual-LLM methods FastDetectGPT and Binoculars while using roughly half their memory; and it is over 20×faster than the perturbation-based DetectCodeGPT with comparable VRAM.

Table 6: Inference costs of different methods on AI Code Detection Education (Strategy 1).

Detectors	**Time** (ms/sample)	**Peak VRAM** (MB)	**AUROC**
LogP	67.7	13073.26	0.670
GLTR	85.6	13073.26	0.647
LRR	87.8	13073.26	0.517
FastDetectGPT	159.2	26050.99	0.800
Binoculars	119.9	25881.90	0.752
DetectCodeGPT	3251.4	13607.49	0.600
CodeRipple	165.8	13285.90	0.925

Failure Mode Analysis CodeRipple struggles on very short code snippets (see Appendix E for a detailed length sensitivity analysis). Besides, there are two primary failure modes: highly templated human-written code tends to be misclassified as LLM-generated (false positive), while low-quality LLM-generated code tends to be misclassified as human-written (false negative). Representative examples are illustrated in Figure 7 of Appendix E.

5 Conclusion

This work studies training-free detection of LLM-generated code, reveals the inadequacy of global TPS statistics, and establishes local TPS dynamics as a robust basis for distinguishing LLM-generated from human-written code. Building on this insight, we propose CodeRipple, the first wavelet-based, training-free detection framework that characterizes local TPS dynamics through two complementary lenses: the Stationary Wavelet Transform to model fluctuation morphology with translation invariance, and the Discrete Wavelet Transform to quantify cross-scale energy distribution. Evaluated on three challenging benchmarks encompassing diverse programming languages, generating LLMs and sophisticated evasion strategies, CodeRipple consistently outperforms previous methods, demonstrating both strong effectiveness and robust generalizability. Our results suggest that reliable code provenance depends not on aggregate perplexity, but on the fine-grained temporal structure of token-level uncertainty. We expect CodeRipple to inspire more extensive applications of signal-processing methodologies in AI content detection.

Limitations

Despite its strong performance, CodeRipple has several limitations. First, it relies on a proxy LLM to compute TPS. Although our experiments show that CodeRipple is generally robust to the choice of proxy model, its effectiveness may degrade when the proxy closely resembles the code generator, which can result in a flattened TPS that lacks sufficient discriminative dynamics. Second, the method assumes that the input code is sufficiently long and structurally diverse to exhibit discernible local dynamics in TPS, and may struggle on very short or highly templated snippets. Third, CodeRipple combines features from both SWT and DWT, which requires weighting hyperparameters. Although our experiments show that an empirically chosen fixed weight yields strong performance and the method is largely insensitive to this choice, the current setting remains heuristic and lacks a principled optimization strategy, a direction we leave for future work.

Acknowledgments

We would like to thank the reviewers for their insightful and valuable suggestions, which have significantly enhanced the quality of this work. This work is supported by the National Natural Science Foundation of China (62376033 and 62232006) and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM103).

References

Guangsheng Bao, Yanbin Zhao, Zhiyang Teng, Linyi Yang, and Yue Zhang. 2024. Fast-detectGPT: Efficient zero-shot detection of machine-generated text via conditional probability curvature. In *The Twelfth International Conference on Learning Representations*.

Amrita Bhattacharjee, Tharindu Kumarage, Raha Moraffah, and Huan Liu. 2023. ConDA: Contrastive domain adaptation for AI-generated text detection. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 598–610, Nusa Dua, Bali. Association for Computational Linguistics.

Sufiyan Bukhari, Benjamin Tan, and Lorenzo De Carli. 2023. Distinguishing ai- and human-generated code: A case study. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, SCORED '23, page 1725, New York, NY, USA. Association for Computing Machinery.

Souradip Chakraborty, Amrit Bedi, Sicheng Zhu, Bang An, Dinesh Manocha, and Furong Huang. 2024. Position: On the possibilities of AI-generated text detection. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 6093–6115. PMLR.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 34 others. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Yanlong Chen, Mattia Orlandi, Pierangelo Maria Rapa, Simone Benatti, Luca Benini, and Yawei Li. 2025. Physiowave: A multi-scale wavelet-transformer for physiological signal representation. *Preprint*, arXiv:2506.10351.

Ingrid Daubechies. 1988. Orthonormal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 41(7):909–996.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 982–994.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander Rush. 2019. GLTR: Statistical detection and visualization of generated text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 111–116, Florence, Italy. Association for Computational Linguistics.

Alexandre Grossmann and J. Morlet. 1984. Decomposition of hardy functions into square integrable wavelets of constant shape. *SIAM Journal on Mathematical Analysis*, 15:723–736.

Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. 2025. On the effectiveness of large language models in domain-specific code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(3).

Hanxi Guo, Siyuan Cheng, Xiaolong Jin, Zhuo Zhang, Kaiyuan Zhang, Guanhong Tao, Guangyu Shen, and Xiangyu Zhang. 2024. Biscope: Ai-generated text detection by checking memorization of preceding tokens. In *Advances in Neural Information Processing Systems*, volume 37, pages 104065–104090. Curran Associates, Inc.

Abhimanyu Hans, Avi Schwarzschild, Valeriia Cherepanova, Hamid Kazemi, Aniruddha Saha,

Micah Goldblum, Jonas Geiping, and Tom Goldstein. 2024. Spotting LLMs with binoculars: Zero-shot detection of machine-generated text. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 17519–17537. PMLR.

Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Mei Nagappan. 2024. Whodunit: Classifying code as human authored or gpt-4 generated - a case study on codechef problems. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 394406, New York, NY, USA. Association for Computing Machinery.

Yafu Li, Qintong Li, Leyang Cui, Wei Bi, Longyue Wang, Linyi Yang, Shuming Shi, and Yue Zhang. 2023. Deepfake text detection in the wild. *ArXiv*, abs/2305.13242.

Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*, 50(6):1548–1584.

S.G. Mallat. 1989. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693.

Luca Masserano, Abdul Fatir Ansari, Boran Han, Xiyuan Zhang, Christos Faloutsos, Michael W. Mahoney, Andrew Gordon Wilson, Youngsuk Park, Syama Rangapuram, Danielle C. Maddix, and Yuyang Wang. 2024. Enhancing foundation models for time series forecasting via wavelet-based tokenization. *Preprint*, arXiv:2412.05244.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In *International Conference on Machine Learning*.

G. P. Nason and B. W. Silverman. 1995. *The Stationary Wavelet Transform and some Statistical Applications*, pages 281–299. Springer New York, New York, NY.

Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2024. Gptsniffer: A codebert-based classifier to detect source code written by chatgpt. *J. Syst. Softw.*, 214(C).

Daniil Orel, Dilshod Azizov, and Preslav Nakov. 2025. CoDet-m4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10570–10593, Vienna, Austria. Association for Computational Linguistics.

Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. 2024. Assessing ai detectors in identifying ai-generated code: Implications for education. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24, page 111, New York, NY, USA. Association for Computing Machinery.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of github copilots code contributions. *Commun. ACM*, 68(2):96105.

Joshua S. Richman and Joseph Randall Moorman. 2000. Physiological time-series analysis using approximate entropy and sample entropy. *American journal of physiology-heart and circulatory physiology*, 278 6:H2039–49.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. 2025. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1628–1639.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, Miles McCain, Alex Newhouse, Jason Blazakis, Kris McGuffie, and Jasmine Wang. 2019. Release strategies and the social impacts of language models. *Preprint*, arXiv:1908.09203.

Jinyan Su, Terry Zhuo, Di Wang, and Preslav Nakov. 2023. DetectLLM: Leveraging log rank information for zero-shot detection of machine-generated text. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12395–12412, Singapore. Association for Computational Linguistics.

Eduard Tulchinskii, Kristian Kuznetsov, Laida Kushnareva, Daniil Cherniavskii, Sergey Nikolenko, Evgeny Burnaev, Serguei Barannikov, and Irina Piontkovskaya. 2023. Intrinsic dimension estimation for robust detection of ai-generated texts. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Vivek Verma, Eve Fleisig, Nicholas Tomlin, and Dan Klein. 2024. Ghostbuster: Detecting text ghostwritten by large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics:*

Human Language Technologies (Volume 1: Long Papers), pages 1702–1717, Mexico City, Mexico. Association for Computational Linguistics.

Jinwei Xu, He Zhang, Yanjing Yang, Lanxin Yang, Zeru Cheng, Jun Lyu, Bohan Liu, Xin Zhou, Alberto Bacchelli, Yin Kia Chiam, and Thiam Kian Chiew. 2025. One size does not fit all: Investigating efficacy of perplexity in detecting llm-generated code. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Han Yu, Peikun Guo, and Akane Sano. 2025. Adawavenet: Adaptive wavelet network for time series analysis. *Preprint*, arXiv:2405.11124.

Zhiyuan Yu, Yuhao Wu, Ning Zhang, Chenguang Wang, Yevgeniy Vorobeychik, and Chaowei Xiao. 2023. CodeIPPrompt: Intellectual property infringement assessment of code language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 40373–40389. PMLR.

A Daubechies Wavelets

Daubechies wavelets (Daubechies, 1988) are defined by a pair of finite-length filters: a high-pass wavelet filter $\mathbf{g} = (g_1, \ldots, g_p)$ and a low-pass scaling filter $\mathbf{h} = (h_1, \ldots, h_p)$, where the length $p = 2J$ is determined by the wavelet order J (e.g., the db4 wavelet has $J = 4$ and $p = 8$). The two filters satisfy the quadrature mirror relationship:

$$g_k = (-1)^k \, h_{p+1-k}, \quad k = 1, 2, \ldots, p.$$

The wavelet filter $\mathbf{g}$ is designed to have J vanishing moments, i.e.,

$$\sum_{k=1}^{p} g_k \, (k-1)^m = 0, \quad m = 0, 1, \ldots, J-1.$$

This property ensures zero output when convolved with any polynomial signal of degree less than J, making the filter sensitive only to abrupt fluctuations. Moreover, the associated wavelet function has compact support over an interval of length p, which localizes the analysis to short token spans and aligns with our focus on local TPS dynamics.

B Sample Entropy

Sample entropy (Richman and Moorman, 2000) quantifies the regularity of a time series by measuring the negative logarithm of the conditional probability that two sequences similar for m consecutive points remain similar over $m+1$ consecutive points within a tolerance ϵ. Specifically, given a sequence $\mathbf{r} = (r_1, \ldots, r_N)$, we extract m-dimensional template vectors using a sliding window, yielding overlapping subsequences:

$$\mathbf{r}_i^{(m)} = (r_i, \ldots, r_{i+m-1}), \quad i = 1, \ldots, N-m+1.$$

Pairwise distances are computed using the Chebyshev norm:

$$d(\mathbf{r}_i^{(m)}, \mathbf{r}_j^{(m)}) = \max_{0 \le k < m} |r_{i+k} - r_{j+k}|.$$

Let $A^m(\epsilon)$ denote the number of distinct pairs (i, j) with $i \neq j$ such that $d(\mathbf{r}_i^{(m)}, \mathbf{r}_j^{(m)}) \le \epsilon$, and let $A^{m+1}(\epsilon)$ be the corresponding count for $(m+1)$-dimensional templates. The sample entropy is then defined as

$$\mathrm{SE}(m, \epsilon) = -\log\left(\frac{A^{m+1}(\epsilon)}{A^m(\epsilon)}\right),$$

provided $A^m(\epsilon) > 0$. Following standard practice, we set $m = 3, \epsilon = 0.2\,\sigma_{\mathbf{r}}$, where $\sigma_{\mathbf{r}}$ is the standard deviation of sequence $\mathbf{r}$.

C Details about Weighting Strategies

SWT Weighting Schemes We evaluate 9 weighting schemes for aggregating multi-scale SWT features. The first three are:

- Equal-weight: $\beta_1 : \beta_2 : \beta_3 : \beta_4 = 1 : 1 : 1 : 1$, where all SWT scales contribute equally.
- Small-scale-dominant: $\beta_1 : \beta_2 : \beta_3 : \beta_4 = 4 : 3 : 2 : 1$, which emphasizes fine-grained TPS fluctuations captured by smaller-scale SWT components.
- Large-scale-dominant: $\beta_1 : \beta_2 : \beta_3 : \beta_4 = 1 : 2 : 3 : 4$, which assigns greater importance to larger-scale SWT components that capture coarse-grained TPS fluctuations.

We further introduce six two-scale-dominant configurations, where two SWT scales receive higher relative weights and the other two receive lower ones. Specifically, the weight ratios are defined as:

$$(\beta_1 : \beta_2 : \beta_3 : \beta_4) \in \left\{ \begin{matrix} (7{:}7{:}3{:}3), (7{:}3{:}7{:}3), \\ (7{:}3{:}3{:}7), (3{:}7{:}7{:}3), \\ (3{:}7{:}3{:}7), (3{:}3{:}7{:}7) \end{matrix} \right\}.$$

All weighting schemes satisfy the constraint $\alpha + \sum_{\ell=1}^{4} \beta_\ell = 1$ with α fixed to 0.01.

Table 7: AUROC of CodeRipple under different SWT weighting schemes on AI Code Detection Education dataset.

β's	Evasion Strategies													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**AVG**
1	0.925	0.869	0.879	0.921	0.766	0.867	0.803	0.865	0.875	0.910	0.936	0.943	0.554	0.855
2	0.925	0.871	0.881	0.922	0.786	0.874	0.809	0.860	0.871	0.908	0.937	0.943	0.568	0.858
3	0.921	0.864	0.874	0.918	0.740	0.857	0.794	0.867	0.875	0.908	0.933	0.939	0.537	0.848
4	0.925	0.870	0.880	0.922	0.787	0.872	0.808	0.861	0.872	0.909	0.937	0.943	0.566	0.858
5	0.924	0.870	0.880	0.922	0.762	0.869	0.803	0.863	0.872	0.906	0.936	0.942	0.557	0.854
6	0.922	0.868	0.878	0.920	0.757	0.865	0.803	0.866	0.874	0.910	0.934	0.940	0.541	0.852
7	0.925	0.868	0.878	0.921	0.770	0.867	0.801	0.862	0.874	0.906	0.937	0.944	0.565	0.855
8	0.923	0.866	0.875	0.919	0.766	0.863	0.800	0.866	0.875	0.911	0.935	0.941	0.548	0.853
9	0.921	0.865	0.875	0.919	0.739	0.859	0.795	0.866	0.875	0.907	0.933	0.940	0.540	0.849

Table 8: AUROC of CodeRipple under different DWT weights on AI Code Detection Education dataset.

α	Evasion Strategies													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**AVG**
0.001	0.918	0.860	0.869	0.916	0.756	0.874	0.828	0.836	0.845	0.889	0.933	0.932	0.502	0.843
0.005	0.926	0.871	0.880	0.923	0.777	0.877	0.820	0.853	0.863	0.904	0.939	0.943	0.536	0.855
0.010	0.925	0.871	0.881	0.922	0.786	0.874	0.809	0.860	0.871	0.908	0.937	0.943	0.568	0.858
0.050	0.903	0.844	0.858	0.902	0.783	0.846	0.759	0.858	0.872	0.897	0.915	0.923	0.641	0.846
0.100	0.893	0.832	0.848	0.893	0.776	0.834	0.740	0.853	0.868	0.891	0.906	0.915	0.653	0.839
0.250	0.886	0.822	0.839	0.886	0.771	0.824	0.725	0.849	0.864	0.887	0.899	0.909	0.660	0.832
0.400	0.884	0.819	0.837	0.884	0.769	0.821	0.720	0.847	0.863	0.885	0.897	0.907	0.662	0.830
0.550	0.883	0.817	0.836	0.883	0.768	0.820	0.718	0.847	0.863	0.885	0.896	0.907	0.662	0.830
0.700	0.882	0.816	0.835	0.883	0.768	0.819	0.717	0.846	0.862	0.884	0.895	0.906	0.663	0.829
0.850	0.882	0.816	0.834	0.882	0.767	0.818	0.716	0.846	0.862	0.884	0.895	0.906	0.663	0.829

DWT Weighting Schemes We evaluate 10 distinct settings for the DWT weight, i.e., $\alpha \in \{$ 0.001, 0.005, 0.010, 0.050, 0.100, 0.250, 0.400, 0.550, 0.700, 0.850$\}$. In each setting, the relative weights of the SWT scales are fixed to $\beta_1 : \beta_2 : \beta_3 : \beta_4 = 4 : 3 : 2 : 1$, with $\alpha + \sum_{\ell=1}^{4} \beta_\ell = 1$.

Detailed Results for Varying Weighting Schemes Table 7 presents the detailed results of CodeRipple under the nine SWT weighting schemes. The results show that CodeRipple achieves consistently strong performance across all nine settings, demonstrating its insensitivity to the specific choice of SWT scale weights. Moreover, among these configurations, small-scale-dominant weightings, such as $(\beta_1 : \beta_2 : \beta_3 : \beta_4) = (4 : 3 : 2 : 1)$ and $(7 : 7 : 3 : 3)$, yield slightly better performance, suggesting that emphasizing smaller-scale SWT components, which capture fine-grained TPS fluctuations, is more effective for distinguishing LLM-generated code from human-written code. Table 8 further presents the detailed results under the ten DWT weighting settings. The results show that CodeRipple's performance remains relatively stable across different α values, achieving the best result at $\alpha = 0.01$.

D Details about Other Hyperparameters

Number of Wavelet Scales We evaluate $L \in \{1, 2, 3, 4\}$ wavelet scales, using a fixed weighting scheme across all settings: $\alpha = 0.01$ for the DWT component, and scale-wise SWT weights $\beta_1 : \beta_2 : \beta_3 : \beta_4 = 4 : 3 : 2 : 1$ with $\alpha + \sum_{\ell=1}^{4} \beta_\ell = 1$. For a given L, only β_1 to β_L are used in aggregation, without renormalization. Table 9 shows the detail results, validating that performance improves with L, supporting our choice of $L = 4$ as a trade-off between effectiveness and stability.

Subsequence Length for Sample Entropy We evaluate three subsequence lengths $m \in \{1, 2, 3\}$ for sample entropy computation, with results reported in Table 10. The performance of CodeRipple is insensitive to the choice of m, e.g., $m = 2$ and $m = 3$ yield nearly identical results. It is worth noting that when $m \geq 4$, the number of matched subsequences drops sharply, and unmatched cases frequently occur in practice, leading to unstable sample entropy estimates for finite-length TPS.

E Length Sensitivity and Failure Cases

We conduct a fine-grained analysis of CodeRipple's detection performance versus code length on the

Table 9: AUROC of CodeRipple under different numbers of wavelet scales on AI Code Detection Education dataset.

L	Evasion Strategies													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**AVG**
1	0.873	0.828	0.832	0.869	0.769	0.816	0.747	0.831	0.858	0.834	0.876	0.885	0.598	0.817
2	0.899	0.851	0.854	0.892	0.799	0.841	0.781	0.841	0.867	0.873	0.904	0.911	0.593	0.839
3	0.914	0.863	0.868	0.909	0.773	0.856	0.797	0.845	0.863	0.891	0.921	0.927	0.558	0.845
4	0.925	0.871	0.881	0.922	0.786	0.874	0.809	0.860	0.871	0.908	0.937	0.943	0.568	0.858

Table 10: AUROC of CodeRipple under different m in sample entropy on AI Code Detection Education dataset.

m	Evasion Strategies													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**AVG**
1	0.911	0.847	0.860	0.911	0.748	0.869	0.793	0.826	0.839	0.896	0.927	0.933	0.502	0.836
2	0.922	0.868	0.878	0.922	0.771	0.878	0.810	0.853	0.864	0.906	0.935	0.941	0.562	0.855
3	0.925	0.871	0.881	0.922	0.786	0.874	0.809	0.860	0.871	0.908	0.937	0.943	0.568	0.858

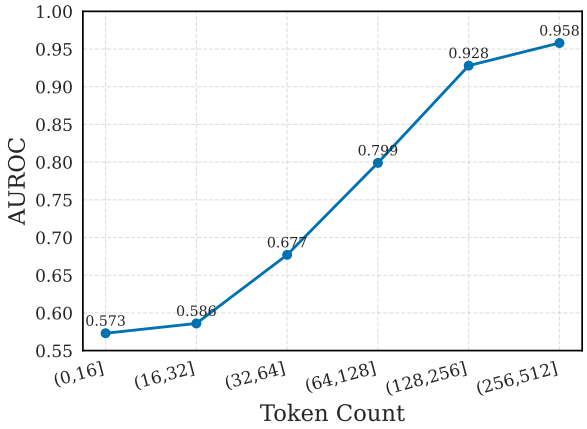

Figure 6: AUROC of CodeRipple stratified by token count on AI Code Detection Education (Strategy 1).

AI Code Detection Education dataset (Strategy 1: unmodified LLM-generated code). Specifically, we categorize all samples into six non-overlapping intervals based on their token counts: (0,16], (16,32], (32,64], (64,128], (128,256], and (256,512], and compute the AUROC for each interval. The results in Figure 6 suggest that a length of 64 tokens effectively delineates the "danger zone" for detection. Performance remains constrained for code snippets under 64 tokens (AUROC < 0.68), enters a usable regime within the (64, 128] range (AUROC$\approx$0.80), and achieves high reliability (AUROC > 0.92) once the length exceeds 128 tokens.

Figure 7 showcases two representative failure cases. The first involves highly templated human code misidentified as LLM-generated, whereas the second depicts low-quality LLM output misclassified as human-written, reflecting the inherent challenges in detecting structural extremes.

Case 1: Human (GT) → LLM (Pred)

```
class Solution:
    def firstBadVersion(self, n):
        """
        :type n: int
        :rtype: int
        """
        left, right = 0, n
        if isBadVersion(1):
            return 1
        while left < right:
            middle = (left + right) // 2
            if isBadVersion(middle):
                right = middle
            else:
                left = middle + 1
        return left
```

Case 2: LLM (GT) → Human (Pred)

```
import pandas as pd

df = pd.read_excel('coalpublic2013.xlsx')
specific_msha_id = df[df['MSHA ID'] ==
↪  desired_msha_id]

    if math.sin(x = math.pi / 2 ) > 1:
        print(""Sine of 90 degrees is
        ↪  less than 1"")

specific_msha_id
```

Figure 7: Representative failure cases on AI Code Detection Education dataset (Strategy 1). Top: a highly templated human-written code snippet misclassified as LLM-generated. Bottom: a low-quality LLM-generated code snippet misclassified as human-written.