USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

PVMark: Enabling Public Verifiability for LLM Watermarking Schemes

Haohua Duan, *East China University of Science and Technology and Shanghai Jiao Tong University;* Liyao Xiang, *Shanghai Jiao Tong University and Shanghai Innovation Institute;* Xin Zhang, *Ant Group;* Baochun Li, *University of Toronto;* Bo Li, *Hong Kong University of Science and Technology*

https://www.usenix.org/conference/usenixsecurity26/presentation/duan

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

PVMark: Enabling Public Verifiability for LLM Watermarking Schemes

Haohua Duan[1,2]*, *Liyao Xiang*[2,3]†, *Xin Zhang*[4], *Baochun Li*[5], *and Bo Li*[6]

[1]*East China University of Science and Technology, China*
[2]*Shanghai Jiao Tong University, China*
[3]*Shanghai Innovation Institute, China*
[4]*Ant Group, China*
[5]*University of Toronto, Canada*
[6]*Hong Kong University of Science and Technology, China*

Abstract

Watermarking schemes for large language models (LLMs) have been proposed to identify the source of the generated text. However, current watermarking solutions hardly resolve the trust issue: the watermark detection often relies on the secret key and thus remains opaque to the public; otherwise any adversary may launch removal attacks if the secret key is exposed. To resolve the dilemma, we propose PVMark, a plugin based on zero-knowledge proof (ZKP), enabling the watermark detection process to be publicly verifiable by third parties without disclosing any secret key. PVMark novelly hinges upon the proof of 'correct execution' of watermark detection on which a set of constraints are built, and is optimized according to the structural nature of the detection process. Developed for three representative watermarking schemes, we implement multiple variants of PVMark in `Python`, `Rust` and `Circom`, covering combinations of three hash functions and four ZKP protocols, showing our approach effectively works under a variety of circumstances. By experimental results, PVMark efficiently enables public verifiability on the state-of-the-art LLM watermarking schemes yet without compromising the watermarking performance, and hence is promising for practical deployment.

1 Introduction

In recent years, large language models (LLMs) have significantly impacted the research community and society by generating high-quality texts akin to human-written content. However, this advancement also introduces various real-world threats, such as the dissemination of generated fake news [37] and the misuse of LLMs in academic misconduct [48]. To ensure the development of responsible AI, tracing the provenance of texts generated by LLMs is essential.

To address these concerns, the research community has proposed various watermarking schemes for LLMs [7, 19, 23, 24, 28, 39, 50, 55, 56, 59]. We focus on the mainstream sampling-based watermarking schemes where the probability distribution of generated tokens is modified based on the context and a secret key held by the owner. The watermark is subsequently extracted by detecting statistical significance within the token distribution of the suspected text. However, such schemes encounter a significant trust deficit in practical applications. As illustrated in Figure 1, when the owner of the LLM asserts rights against unauthorized use, they may face inquiries from entities outside their trust boundary–*e.g.,* online platforms seeking independent moderation evidence, third-party auditors, investigative journalists conducting external reviews, or courts. The legitimacy of the watermark is often questioned due to the opacity of detection procedures.

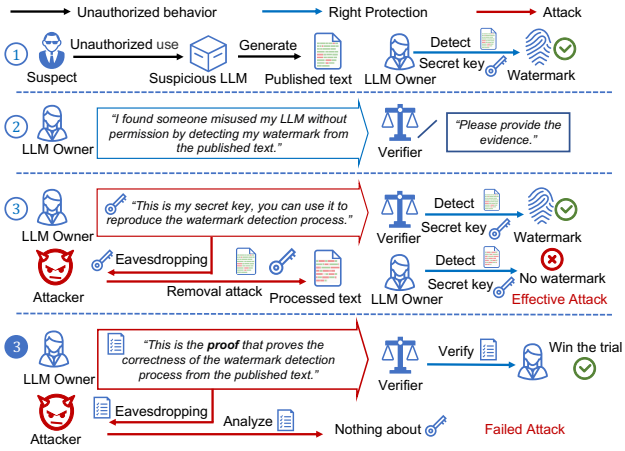

Figure 1: From top to bottom: ① The owner of the LLM detects unauthorized use of the model via watermark detection. ② The owner initiates legal action against the suspected party, and the verifier requests evidence. ③ The secret key of the owner is exposed while responding to the request of the verifier. ❸ The owner provides the proof as evidence under *PVMark*, thereby preventing the attack. Potential verifiers include online platforms seeking independent moderation evidence, third-party auditors, investigative journalists, or courts.

*Most work is done at Shanghai Jiao Tong University.
†Corresponding author: xiangliyao08@sjtu.edu.cn.

In such scenarios, the owner faces pressure to disclose key-related secrets for third-party reproduction. Conversely, the verifier often resides outside a strict confidentiality boundary, which increases the risk of eavesdropping (or accidental leakage) that could compromise the security of the watermark. The root cause of this problem is the reliance on a secret key that cannot be made public (as this exposes the watermarked model to removal or ambiguity attacks), nor kept strictly private (as this fails to convince others of the validity of the detection). Consequently, existing LLM watermarking schemes lack *public verifiability*.

One might argue that confidentiality can be managed via legal mechanisms such as non-disclosure agreements (NDAs) or protective orders. However, real-world cases indicate that procedural safeguards are often insufficient, as the disclosure of highly sensitive technical secrets may still be compelled or leak to unauthorized parties once they leave the boundary of the owner [2, 16]. A potential engineering mitigation is key rotation [1], i.e., disclosing the key required for each verification. However, due to limited storage, watermark key sequence [27] is used in practice by periodically cycling keys. This ultimately threatens the generated contents sharing the same secret key (and context) with a previously disputed text. Therefore, cryptographic *public verifiability* is required: the ability to convince arbitrary third parties that watermark detection was performed correctly, without revealing the key.

Prior research has attempted to resolve this issue. Fairoze *et al.* propose using a private key for watermark embedding and a public key for detection [10]. However, their detection mechanism relies heavily on extracted features of the generated texts, which are susceptible to exploitation by adversaries. Liu *et al.* address the problem by training two separate neural networks as watermark embedding and detection modules, respectively [29]; nevertheless, the exposure of the detection network enables an adversary to train an inverse network to remove the watermark. Kirchenbauer *et al.* propose a private detection scheme that restricts access to detection APIs to authorized parties [23], yet this approach fails to prove the integrity of the API execution to external verifiers.

To bridge this gap, we introduce a mechanism named *PVMark*, which augments state-of-the-art LLM watermarking schemes with *public verifiability* while preserving the effectiveness, fidelity, and robustness of the original schemes. *PVMark* achieves public verifiability by integrating zero-knowledge proofs (ZKP) into watermark detection. ZKP is a protocol in which a prover can convince a verifier of the truth of a statement without conveying any information beyond the validity of that statement. This aligns with our objective of verifying that watermark detection is correctly performed without revealing the secret key.

To instantiate this concept, we select three representative LLM watermarking schemes–KGW [23], SynthID-Text [7], and Segment-Watermark [39]–to implement *PVMark*. We observe that these schemes share a common token-sampling-alteration framework: at each sampling round, all candidate tokens are randomly divided into groups based on the secret key and the watermark. Tokens in the 'green' group are then assigned a higher probability of being sampled as the output. Consequently, the statistical attributes of the output is collected to extract the watermark.

Our core idea is to provide evidence that the statistical results in the produced text are faithfully collected without exposing the secret key, thereby demonstrating that the text indeed contains watermarks. Two criteria must be satisfied: the watermark is extracted from the text, and the proof of the detection passes verification. In applying ZKP to watermarking, we primarily address two technical challenges: 1) modeling the watermark detection by circuits on which ZKP protocols operate, and 2) systematically optimizing the design to improve the proving/verification efficiency.

To address the first challenge, we decompose the watermark detection into four components for reconstruction: mapping table check, hashing, comparison, and summation. We replace the original components with ZKP-friendly alternatives. For instance, the pseudorandom function (PRF) instantiated by a linear congruential generator based on modular operations is inefficient for ZKP; therefore, it is replaced with cryptographic hash functions. The identification of 'green' tokens is reconfigured as a comparison of the generated random numbers against a fixed threshold. We demonstrate that this renovation of watermarking schemes not only preserves the properties of the watermark but also tailors them to ZKP protocols.

In addition to reconstruction, we propose multiple approaches to enhance the efficiency of *PVMark* (addressing the second challenge). Since hash functions are utilized extensively in detection, we combine them into three-to-one hash functions to create more succinct constraints. Observing that watermark detection is a structural procedure over a sequence of tokens, we treat the detection of a token subsequence as an instance and reduce the checking of multiple instances to a single check by folding the proofs [25]. This reduces the ZKP overhead for proving multiple instances to proving one final instance and the folding steps, significantly mitigating the computational cost of *PVMark*.

Highlights of our contributions are as follows. We propose *PVMark* that enables *public verifiability* on zero- and multi-bit LLM watermarking schemes, achieving a compromise between public detection and authorized detection, resolving the trust issue in watermarking detection. *PVMark* is built upon ZKP techniques and incorporates application-specific optimizations. Implementation-wise, we realize multiple variants of *PVMark* on `Python`, `Rust` and `Circom`, including different combinations of three LLM watermarking schemes, three hash functions, and four ZKP protocols. The security analysis and experimental results have demonstrated that *PVMark* is applicable to be deployed in LLM watermarking service with low costs for public verification.

2 Related Work

To identify and trace LLM-generated text, several schemes have been proposed. One approach is retrieval-based schemes [26], which require storing all generated texts, thereby compromising user privacy. Another approach is post hoc detection schemes [18, 35, 49], which use machine learning models to distinguish LLM-generated text from human-written text. However, these schemes require significant computational resources and have limited performance [9]. A third approach is LLM watermarking, represented by the KGW scheme [23], which divides the vocabulary into green and red token lists and adjusts logits to shift the token distribution toward green tokens. This method enables watermarking without LLM fine-tuning, making it more general and cost-effective.

Subsequent probability-shifting-based schemes include Lee *et al.* [28] and Wang *et al.* [50], which reduce logits modification impact by watermarking high-entropy tokens. Yoo *et al.* [56] propose assigning different message bits to distinct text positions to increase watermark payload. Qu *et al.* use dynamic programming for balanced token allocation, improving watermark detection accuracy. Kirchenbauer *et al.* [24] study hash strategies to enhance robustness. Zhao *et al.* [59] show that globally fixed vocabulary partitioning improves resistance to removal attacks. Hu *et al.* [19] and Wu *et al.* [55] propose unbiased weighting to enhance imperceptibility in KGW. Dathathri *et al.* [7] introduce the Tournament sampling algorithm to improve watermark detection while maintaining text quality.

However, these schemes are mainly suitable for private watermark detection and cannot support public detection, as exposing the detector introduces various threats.

To address public verifiability, Fairoze *et al.* [10] use asymmetric encryption for embedding the watermark with a private key and detecting it with a public key, but this relies on text features, which can be exploited by adversaries for spoofing attacks. Liu *et al.* [29] propose training two neural networks, one for embedding and one for detection. However, when the window size is small, this method is vulnerable to reverse engineering, where adversaries could reverse train the embedding network from the detection network, compromising robustness.

3 Preliminaries

3.1 Zero Knowledge Proof

A zero-knowledge proof (ZKP) protocol is a cryptographic method enabling a prover to convince a verifier that a statement is true using private input w and public input u, satisfying completeness, soundness, and zero-knowledge properties.

ZKP protocols operate on arithmetic circuits that encode computations as constraints, e.g., GKR-based arithmetizations, rank-1 constraint systems (R1CS), PLONKish arithmetizations, and related arithmetizations. For notation, a standard PLONKish-style constraint is used:

$$s_L \cdot x_L + s_R \cdot x_R + s_O \cdot x_O + s_M \cdot x_L \cdot x_R + s_C \cdot c = 0, \quad (1)$$

where selectors $s_L, s_R, s_O, s_M, s_C \in \mathbb{F}$ parameterize operations and $x_L, x_R, x_O, c \in \mathbb{F}$ are wire/constant values.

Recursive ZKP. Nova [25] is a folding-based recursive ZKP that realizes incrementally verifiable computation (IVC):

$$u_\eta = F(w_{\eta-1}, F(w_{\eta-2}, F(\ldots F(w_1, F(w_0, u_0))))) \quad (2)$$

where F, w, u and η denote the computation function, private input, public input and the number of computations, respectively. Nova folds two relaxed R1CS instances into one; a relaxed R1CS instance is:

$$(\mathsf{A} \cdot \mathsf{Z}) \odot (\mathsf{B} \cdot \mathsf{Z}) = \mu(\mathsf{C} \cdot \mathsf{Z}) + \nu \quad (3)$$

where $\mathsf{A}, \mathsf{B}, \mathsf{C}$ are matrices containing ZKP circuits information, μ is a scalar and ν is used to store redundant items. The $\odot$ denotes the hadamard product. $\mathsf{Z} = [u, w, 1]$ is the instance vector including the witness w, public input u and the constant 1. Two relaxed R1CS instance $(\mathsf{Z}_1, \mu_1, \nu_1)$ and $(\mathsf{Z}_2, \mu_2, \nu_2)$ can be folded into one instance (Z, μ, ν) as follows:

$$\begin{aligned} \nu =& \nu_1 + r \cdot (\mathsf{A}\mathsf{Z}_1 \odot \mathsf{B}\mathsf{Z}_2 + \mathsf{A}\mathsf{Z}_2 \odot \mathsf{B}\mathsf{Z}_1 \\ & - \mu_1 \cdot \mathsf{C}\mathsf{Z}_2 - \mu_2 \cdot \mathsf{C}\mathsf{Z}_1) + r^2 \cdot \nu_2, \quad (4) \\ \mu =& \mu_1 + r \cdot \mu_2, \quad (5) \\ \mathsf{Z} =& \mathsf{Z}_1 + r \cdot \mathsf{Z}_2, \quad (6) \end{aligned}$$

where r denotes randomness. Proving the folded instance and the folding relations suffices, potentially reducing constraints compared to proving all instances separately.

3.2 Language Model Watermarking

Given a prompt $x = [x_0, \ldots, x_{m-1}]$, a language model $\mathcal{M}$ iteratively computes the logit scores $l_j^{(i)}$ $(0 \le i \le n-1, j \in |V|)$ of the next n tokens over the vocabulary V. These logits are transformed into a probability distribution $p^{(i)}$ by which the model $\mathcal{M}$ samples the next n tokens $y = [y_0, \ldots, y_{n-1}]$. The notation y_i is overloaded to denote the vocabulary index of the sampled token. Language model watermarking embeds a watermark into the text y by altering the token distributions into $\hat{p}^{(i)}$ before sampling. Formally,

Definition 3.1. *A language model watermarking scheme consists of two probabilistic polynomial-time (PPT) algorithms (Watermark Embedding, Watermark Detection):*

- *Watermark Embedding $E(ek, \mathcal{M}) \to dk, y$: the algorithm uses an embedding key ek which embeds watermark into y. Additionally, it returns a detection key dk associated with ek.*

Table 1: The watermark embedding process of zero-bit and multi-bit LLM watermarking schemes. Steps requiring adjustment for PVMark are marked in yellow. PRF$(\cdot)$ denotes a pseudo-random function.

	Step 1: Random seed generation.	**Step 2**: Vocabulary partitioning.	**Step 3**: Probability shifting and sampling.
KGW [23]	Compute random seed $sd_i \leftarrow \text{PRF}(sk, y_{\Psi})$. $\Rightarrow$ Compute random seed $sd_i \leftarrow \text{Hash}(sk, y_{\Psi})$.	a) Assign random number $g^{(y_j)} = \text{PRF}(sd_i, y_j)$ to each token $y_j \in V$; $\Rightarrow$ Assign random number $g^{(y_j)} = \text{Hash}(sd_i, y_j)$ to each token $y_j \in V$; b) Sort V by all the random numbers $g^{(y_1)}, \ldots, g^{(y_{\lvert V\rvert})}$ corresponding to token $y_1, \ldots, y_{\lvert V\rvert} \in V$ and designate the top $\gamma\lvert V\rvert, \gamma \in (0,1)$ tokens in the sorted V as green list G. $\Rightarrow$ Partition the all candidate tokens as follows: $$\text{Candidate token } y_j \in \begin{cases} G, & g^{(y_j)} < \gamma \cdot \lvert\mathbb{F}\rvert \\ R, & g^{(y_j)} \geq \gamma \cdot \lvert\mathbb{F}\rvert \end{cases}$$ for $j \in \lvert V\rvert$, where R and $\mathbb{F}$ denotes the red list and finite field, respectively.	Add a bias δ to logits of all tokens in G and compute the new probability distribution $\hat{p}^{(i)}$ using new logits over V. Sample y_i from $\hat{p}^{(i)}$ and append y_i to y.

SynthID-Text [7]		Generate top K candidate set V for this round and assign ξ random numbers $g_k^{(y_j)} = \text{PRF}(sd_i, y_j, k)$ to each token $y_j \in V$ for $k \in \{1, \ldots, \xi\}$; $\Rightarrow$ Generate top $K = 2^{\xi}$ candidate set V for this round and assign ξ random numbers to each token $y_j \in V$ by
		$$g_k^{(y_j)} = \begin{cases} 1, & \text{Hash}(sd_i, y_j, k) < \lvert\mathbb{F}\rvert/2, \\ 0, & \text{Hash}(sd_i, y_j, k) \geq \lvert\mathbb{F}\rvert/2, \end{cases}$$
		for $k \in \{1, \ldots, \xi\}$;
		For each $y_j \in V$, set $y_j^{(0)} \leftarrow y_j$.
		for $k = 1, \ldots, \xi$ **do**
		  for $j = 0, \ldots, 2^{\xi-k} - 1$ **do**
		  $J = [y_{2j}^{(k-1)}, y_{2j+1}^{(k-1)}]$ (may contain repeats).
		  $J^* = [y \in J \ s.t. \ g_k^{(y)} = \max_{y' \in J} g_k^{(y')}]$ (may contain repeats).
		  Sample $y_j^{(k)}$ from the uniform distribution of J^*.
		  end for
		end for
		Append y_0^{ξ} to y.	

Segment-Watermark [39]	a) Find position index $p_i = M[y_{\Psi}]$; b) Compute random seed $sd_i \leftarrow \text{PRF}(sk, y_{\Psi}, msg[p_i])$. $\Rightarrow$ Compute random seed $sd_i \leftarrow \text{Hash}(sk, y_{\Psi}, msg[p_i])$.	The same as **KGW**.	The same as **KGW**.

- *Watermark Detection $D(dk, y) \rightarrow \{0,1\}$: it takes the detection key dk and a text sequence y as the input, and returns 1 indicating y was indeed generated by $\mathcal{M}$, or 0 otherwise.*

LLM watermarking frameworks. Probability-shifting LLM watermarking schemes can be broadly categorized into *zero-bit* schemes [7, 19, 23, 24, 28, 50, 55, 56, 59] and *multi-bit* schemes [39]. Zero-bit schemes aim to distinguish LLM-generated text from human text without embedding explicit messages, whereas multi-bit schemes embed metadata (*e.g.,* owner identity) for finer attribution.

Despite differences in instantiation, both categories follow a shared high-level framework; representative schemes considered here include the zero-bit schemes KGW [23] and SynthID-Text [7], and the multi-bit scheme Segment-Watermark [39].

The watermark embedding process is summarized in Table 1 and can be divided into three steps: random seed generation, vocabulary partitioning, and probability shifting and sampling. Particularly for SynthID-Text, the latter two steps are indistinguishable.

In each round i, a seed sd_i is derived from the secret key sk and recent context y_{Ψ} (window width Ψ). For multi-bit schemes, tokens are additionally mapped to positions p and the corresponding message bit $msg[p]$ is incorporated into seed generation. The seed then determines a token grouping, and sampling is biased toward the selected group (*e.g.,* via logit bias), thereby embedding watermark signal into the output. In SynthID-Text, tournament sampling compares candidate g-values over multiple rounds and implicitly increases the selection probability of the winner.

The watermark detection processes for zero-bit and multi-bit schemes exhibit subtle distinctions, shown in Table 2. For zero-bit schemes, the seed induced by (sk, y_{Ψ}) determines the grouping per round; the detector counts selected-group tokens $\lvert y\rvert_G$, computes a score $\text{Score}(y)$, and compares it against a threshold τ. For multi-bit schemes, the detector evaluates statistics under all candidate message values (together with sk and context) and outputs the message achieving the highest statistic as the decoded watermark.

4 Problem Statement and PVMark Overview

This section summarizes the threats faced by common LLM watermarking frameworks in content moderation and provenance tracing and overviews PVMark.

Threat model. In content moderation and provenance tracing, the watermarking frameworks involve a watermark embedding party (typically the model owner) and a watermark detection party that extracts watermarks from a suspect text as evidence of origin. Prior schemes [7, 19, 23, 24, 28, 39, 50, 55, 56, 59] typically assume the detector is run by the embedding party, since it holds the detection key. This confidentiality, however, *undermines public credibility*: external verifiers cannot reproduce the detection without access to secrets. If a third party runs detection, it must keep the detection key confidential; any leakage enables adaptive *removal attacks*.

Why exposure of key leads to de-watermarking attacks. The vulnerability stems from the secret key sk used in both

Table 2: The watermark detection process of zero-bit and multi-bit LLM watermarking schemes. Steps requiring adjustment for PVMark are marked in yellow.

	Step 1: Compute random seed	**Step 2**: Check and count green tokens	**Step 3**: Compute score / message
KGW [23]	Compute random seed $sd_i \leftarrow \text{PRF}(sk, y_\psi)$. ⇒ Compute random seed $sd_i \leftarrow \text{Hash}(sk, y_\psi)$.	a) Similar to Step 2 of the watermark embedding process, assign a random number to each token in the vocabulary using sd_i and reorder the vocabulary to find G_i; ⇒ Compute the random number $g^{y_i} = \text{Hash}(sd_i, y_i)$ corresponding to the current token y_i; b) Count the number of green tokens $\lvert y\rvert_G \mathrel{+}= \mathbb{1}(y_i \in G_i)$ where $\mathbb{1}(\cdot)$ is the indicator function. ⇒ Count the number of green tokens $\lvert y\rvert_G \mathrel{+}= \mathbb{1}(g^{y_i} < \gamma \cdot \lvert\mathbb{F}\rvert)$.	$\text{Score}(y) = (\lvert y\rvert_G - \gamma n)/\sqrt{n\gamma(1-\gamma)}$
SynthID-Text [7]		a) Compute ξ random number $g_k^{y_i}$ corresponding to the current token y_i where $k \in \{1, \ldots, \xi\}$;⇒ Compute ξ random number corresponding to the current token y_i by $g_k^{(y_i)} = \begin{cases} 1, & \text{Hash}(sd_i, y_i, k) < \lvert\mathbb{F}\rvert/2, \\ 0, & \text{Hash}(sd_i, y_i, k) \geq \lvert\mathbb{F}\rvert/2, \end{cases}$ for $k \in \{1, ..., \xi\}$; b) Update S_g: $S_g \mathrel{+}= \sum_{k=1}^{\xi} g_k$.	$\text{Score}(y) = \frac{S_g}{(n-\psi)\cdot\xi}$

Segment-Watermark [39]	a) Find position index $p_i = M[y_\psi]$;
	b) Compute all possible random seeds $sd_i^j \leftarrow \text{PRF}(sk, y_\psi, j)$ for $j = 0, 1, \ldots, 2^{\hat{m}} - 1$. ⇒ Compute all possible random seeds $sd_i^j \leftarrow \text{Hash}(sk, y_\psi, j)$ for $j = 0, 1, \ldots, 2^{\hat{m}} - 1$.
		for $j = 0, 1, \ldots, 2^{\hat{m}} - 1$ **do**
		Assign a random number to each token in the vocabulary using sd_i^j and reorder the vocabulary to find G_i^j; ⇒ Compute the random number $g^{y_i} = \text{Hash}(sd_i^j, y_i)$ corresponding to the current token y_i;
		Update COUNT: $\text{COUNT}[p_i][j] \mathrel{+}= \mathbb{1}(y_i \in G_i)$. ⇒ Update COUNT: $\text{COUNT}[p_i][j] \mathrel{+}= \mathbb{1}(g^{y_i} < \gamma \cdot \lvert\mathbb{F}\rvert)$.
		end for
			$msg[p] \leftarrow \arg\max_{j \in [0, 2^{\hat{m}}-1]} \text{COUNT}[p][j]$ for $p = 0, 1, \ldots, \hat{n} - 1$.

embedding and detection. Watermarking derives a per-round seed sd_i from sk and the recent context (Step 1 of Table 1 and Table 2). If sk is revealed, the vocabulary partition (*e.g.*, green/red grouping) becomes predictable, allowing an adversary to selectively replace green tokens and remove the watermark. Even if only sd_i is exposed for a given context, the token-selection rule for the particular context is revealed threatening relevant text. Therefore, both sk and the derived sd_i must remain confidential.

Motivation. The core dilemma is that sk cannot be public, while detection should be transparent to convince arbitrary third parties. *PVMark* resolves this by providing a correctness proof of the watermark detection: by integrating ZKPs, *PVMark* proves that detection was executed faithfully on the given text without disclosing sk (zero-knowledge), with negligible cheating probability (soundness) and successful verification for honest execution (completeness). Such a proof is only expected in post-hoc, disputable cases, e.g., providing supporting evidence in the court, but not as a routine in the watermark embedding and detection pipeline. This is because ZKPs generally are not lightweighted and thus are inapropriate to add to the model generation path. Nevertheless, the watermarking embedding and detection should be efficiently adapted to the post-hoc verification.

Why ZKP. ZKP best matches our goal: public verifiability, non-interactivity, no hardware trust assumption, and detector-side secrecy. Alternatives do not provide the same combination of properties. Message authentication codes and signatures authenticate outputs or identities, but do not prove the correctness proof of a key-based detection algorithm. Commitments and Merkle trees can bind secret values, but public correctness checking requires openings; opening leaks the secret, while hiding the opening prevents third parties from checking the computation. Secure multi-party computation and homomorphic encryption protect participants during computation, but do not naturally leave a succinct public certificate for arbitrary third parties. Trusted execution environments can attest execution, but introduce a hardware trust assumption.

Problem statement. Formally, four parties are involved in *PVMark*: (1) *watermark embedding party* $\mathcal{E}$ who embeds the watermark into $\mathcal{M}$; (2) *watermark detection party* $\mathcal{D}$ who has access to the watermark detector, secret key sk, and text sequence y. $\mathcal{D}$ extracts watermark from y while acting as the *prover* to prove the claim that 'y indeed contains watermark, and the watermark information is *msg* (for multi-bit watermarking);' (3) *verifier* $\mathcal{V}$ who could be any party responsible to verify $\mathcal{D}$'s claim. $\mathcal{V}$ is honest in executing the proof verification while being curious about sk and other credentials that might be contained in the proof provided by $\mathcal{D}$; (4) *adversary* $\mathcal{A}$ who maliciously tampers with y to generate unauthorized text with watermark removed.

Use case: *Content moderation.* A social media platform works with the AI company to detect the AI-generated content (AIGC) on its platform by extracting watermarks from the content. The proof of watermark detection serves as evidence for that the extraction is properly done and the detection is trustworthy. The process does not reveal any key yet preserving moderation credibility.

Provenance tracing. The owner $\mathcal{E}$ generates content y with an embedded watermark. If the origin of y is questioned, $\mathcal{D}$ detects the watermark and provides a verifiable proof to $\mathcal{V}$. $\mathcal{E}$ cannot deny the results backed by this evidence.

In both use cases, embedding and detection algorithms are publicly known, except for the secret key sk, which $\mathcal{D}$ uses in

detection and proof generation without revealing it.

Overview. Formally, we define *PVMark* as follow:

Definition 4.1. *PVMark is a plugin that enables public verifiability on a language model watermarking scheme. It consists of a pair of embedding and detection algorithms:*

- *Watermark embedding WE(sk, $\mathcal{M}$) $\rightarrow$ y: $\mathcal{E}$ uses a secret key sk to produce a watermarked text sequence y.*
- *Watermark detection WD(sk, y) $\rightarrow \{0,1\}$ or msg, Π: $\mathcal{D}$ takes the secret key sk associated with y as inputs. For zero-bit watermarking schemes, it detects whether y contains watermark and returns 1 if present, otherwise 0; for multi-bit watermarking schemes, it detects and returns the watermark information msg embedded in y. Then $\mathcal{D}$ generates a correctness proof Π for the detection.*

Previous works [7, 19, 23, 24, 28, 39, 50, 55, 56, 59] mainly focus on *effectiveness and fidelity*, ensuring high watermark detection accuracy while maintaining text quality, and *robustness*, preventing watermark removal or forgery. PVMark augments watermarking with *public verifiability*, allowing any third party to check the integrity of the detection result without compromising these attributes.

It is noteworthy that Pang *et al.* point out an inherent conflict between robustness and public detectability for watermarking systems in [38]. In contrast, *PVMark* effectively achieves robustness and public verifiability at the same time. *PVMark* is not a publicly detectable scheme, but its watermarking detection process can be verified publicly, adding credibility to the detection.

PVMark does not rely on the specific formulation of the watermark embedding (e.g., red-green formulation) and detection, but hinges upon a more general requirement: the detector can be decomposed into circuit-friendly subroutines whose sensitive inputs is kept as private witnesses. Our implemented instantiations focus on sampling-based probability-shifting schemes since they share a clean seed-to-grouping-to-statistic pipeline. In these schemes, detection can be expressed with hash computation, comparison, lookup or membership checks, and aggregation. Semantic-level [30] or sentence-level watermarking methods [60] is compatible in principle given their easy-to-arithmetize detection logic.

5 Adapting Watermark Schemes for PVMark

Since the original watermark schemes contain procedures that are challenging to construct proofs, we adapt them to almost equivalent, ZKP-friendly forms. The adaptation preserves the original schemes' foundational properties of effectiveness, fidelity, and robustness.

5.1 Hash-Based Vocabulary Partitioning

A key step in watermark embedding is partitioning the vocabulary into green and red lists (Step 2 in Table 1), with green tokens selected at a higher probability than random.

The original watermarking schemes use two methods for vocabulary partitioning. The first is random permutation, used in KGW [23] and Segment-Watermark [39], where all candidate tokens are randomly permuted, and the first $\gamma \cdot |V|$ tokens are marked as green. This method requires a mapping from random numbers to vocabulary indices, involving ZK-unfriendly modulo operations.

The second is the tournament method, adopted by SynthID-Text [7]: it assigns random numbers to each token in the candidate token set by using a precomputed lookup table and a pseudo random function (PRF) instantiated by a linear congruential generator (LCG). Initially, a lookup table of size 2^{16} is randomly populated with 0s and 1s. Then, the LCG generates ξ random numbers, which are mapped to the range of $[0, 2^{16}-1]$ for retrieving values from the table. These values are then assigned to $g_1, ..., g_\xi$. Subsequently, all candidate tokens undergo multiple rounds of competition, where tokens with smaller g values are eliminated iteratively. Tokens that remain in the final round are designated as green tokens. This process also involves ZK-unfriendly modulo operations in LCG.

To avoid modulo operations, we propose a Hash-based method for vocabulary partition. A straightforward and equivalent substitute for random permutation is random numbers sorting where we sort $r = \{r_1, \ldots, r_{|V|}\}$ in numerical order, and take the tokens corresponding to the first $\gamma|V|$ indices in r to compose the green list. Unfortunately, the proof of sorting is still highly inefficient to implement. Hence we take an approximated approach by assigning hash-generated random numbers to candidate tokens. The vocabulary is divided into green/red lists by comparing these numbers with a fixed threshold $\gamma \cdot |\mathbb{F}|$, shown by the first row of Step 2 in Table 1. Similarly, for the tournament method, we compare hash-generated random numbers with a fixed threshold $\frac{|\mathbb{F}|}{2}$, and map them to 0 or 1 depending on the comparison result (shown by the second row of Step 2 in Table 1). However, our hash-based approach relies on the assumption that the hash-generated random numbers are uniformly distributed within the range, so that the fixed threshold could divide the vocabulary in proportion. We validate that the assumption mostly holds true in practice (see Sec. 8.2).

In addition, we instantiate the PRF required for computing the random seed using a hash function. Similarly, all PRFs involved in the watermark detection are also instantiated via hash functions. When checking and counting green tokens, we determine whether the current token is a green token by comparing its corresponding random number with a fixed threshold, as shown in Step 2 of Table 2.

5.2 Selection of the Hash Function

We select cryptographic hash functions as PRFs due to their high-quality pseudorandomness, following NIST SP 800-

90A [4], instead of non-cryptographic algorithms like Linear-feedback shift register [45], Blum Blum Shub [5], Mersenne Twister [32], Xorshift [31], which, though faster, are not ZKP-friendly due to high costs in bitwise operation verification.

Common cryptographic hash functions like SHA256, Keccak256, and BLAKE2, while mapping arbitrary-length inputs to fixed 256-bit outputs, involve many bit operations, leading to high ZKP-related costs. Other hash functions are more ZK-friendly, including MiMC [3], Poseidon [14] and Poseidon2 [15]. These hash functions are effective in finite fields applicable to ZKP and mostly consist of addition and multiplication operations which are relatively efficient to verify. For example, Poseidon hash is based on the sponge function, with a state composed of finite field elements and field operations including addition, multiplication and exponentiation, all of which can be easily converted into ZKP circuits.

To verify the uniformity and randomness of the cryptographic hash functions, we run the chi-square test to examine SHA256, Keccak256, BLAKE2, MiMC, Poseidon and Poseidon2. The results are presented in Sec. 8.2. By tests, we found SHA256, Keccak256, or BLAKE2 are not applicable for *PVMark* since the 256-bit value range of these hash does not match the finite field used by ZKP, and thus the hash values are hardly uniformly distributed on $\mathbb{F}$. MiMC, Poseidon and Poseidon2, hash functions that passed the test, are our final choices for PRFs.

6 Building Blocks of PVMark

To prove watermark detection, *PVMark* generates proofs for the steps in Table 2. In Step 3, the only unknowns are y_G, S_g, and COUNT, while other quantities are public. Therefore, *PVMark* only needs to prove the correctness of y_G, S_g, and COUNT by generating proofs for Step 1 and Step 2, without exposing sk or the green list G. As long as the verifier obtains the correct y_G, S_g, and COUNT, it can derive the score or watermark message to determine whether the given text contains the watermark or extract the embedded watermark. We present the arithmetization of these steps with PLONKish as an example.

6.1 Arithmetize Step 1 in Table 2

Since we have instantiated the PRF by a hash function, we arithmetize the hash to prove Step 1 of Table 2.

Hash functions. The constraints $\mathcal{C}_{hash}(\text{Output};\text{Inputs})$ differ per hash function. We take two common ZK-friendly hash functions, MiMC hash and Poseidon hash, as examples. The MiMC function hashing X is defined as $H_{\text{mimc}}(X) = (f_{\phi-1} \circ f_{\phi-2} \circ \ldots \circ f_0)(X) + k$ where ϕ and k denote the number of rounds and the secret key in MiMC hash, respectively. The round function f_i is defined as $f_i(Y_i) = (Y_i + k + C_i)^{\text{Pow}}$ where C_i is the round constant for round i and $Y_0 = X$. To construct the constraints of $f_i(\cdot)$, we introduce a new power operation selector s_{Pow} to rewrite Eq. (1) as

$$s_{\text{Pow}} \cdot (x_L + x_R + c)^{\text{Pow}} + s_O \cdot x_O = 0, \tag{7}$$

and set $s_{\text{Pow}} = 1, s_O = -1, x_L = Y_i, x_R = k, c = C_i$ and $x_O = f_i(Y_i)$. We construct the MiMC hash by reusing Eq. (7) ϕ times, and use copy constraints to ensure the consistency of input and output in function compositions.

The construction of constraints for Poseidon hash is relatively simpler. Poseidon hash is composed of three components – ARC($\cdot$), S-box($\cdot$), and Mix($\cdot$), for a single round. Specifically, ARC($\cdot$) adds a constant to all inputs, S-box($\cdot$) performs exponentiation wherein the last input of partial round R_p and all inputs of the full round R_f are raised to the power of 5. Similar to Eq. (7), we use s_{Pow} selector to construct the S-box function. In Mix($\cdot$), a pre-calculated matrix is used to perform vector-matrix multiplication operations for all inputs. We also use copy constraints to ensure the consistency of input and output between each round.

Token-position mapping. In the multibit scheme, Step 1 requires looking up the position p for y_ψ from the token-to-position table M. *PVMark* provides a proof for the lookup as it is a part of the counting. However, such a mapping M cannot be disclosed otherwise would allow adversary to infer about the relation between token and the value in $msg[p]$ based on the publicly available COUNT matrix. This would tempt the adversary to remove the watermark from y.

To check the correctness of a set of non-exposable token-position mappings in the watermark detection, we require the prover to precompute the hash values for all possible token-position mappings, use them as leaf nodes to construct a merkle tree, and publish the root value in advance. Thus the proof of a set of mappings reduces to the proof of membership of hash values in the merkle tree.

The membership proof requires the prover to recalculate the root value by the values of the sibling nodes in the path from the given leaf node to the merkle tree root. If the newly calculated root value matches the publicly disclosed one, the given leaf node must exist in the merkle tree. The specific constraints required for membership proof $\mathcal{C}_{mem}$ are as follows:

$$\mathcal{C}_{mem}(y, p, \text{ROOT}) = \begin{cases} H_0 = \mathcal{C}_{hash}(y, p, sk), \\ path_i \cdot (1 - path_i) = 0, \\ H_i = path_i \cdot \mathcal{C}_{hash}(H_{i-1}, \text{Sib}_i) \\ \quad + (1 - path_i) \cdot \mathcal{C}_{hash}(\text{Sib}_i, H_{i-1}), \\ H_L - \text{ROOT} = 0, \end{cases} \tag{8}$$

for $i = \{1, \ldots, L\}$ where y, p, ROOT, sk denote the token index, position index, a publicly known Merkle tree root value, and the secret key, respectively. The $path_i \in \{0, 1\}$ refers to the left/right child of the Merkle tree nodes, which influences the order of hash inputs during computation. Sib_i denotes the value of sibling node along the path from the given leaf node to the merkle tree root.

6.2 Arithmetize Step 2 in Table 2

In Step 2, we compute the random number for token y, compare it with a fixed threshold, and accumulate the result if the token is green. Hence we arithmetize the comparison and summation to prove the step.

Comparison. Assume that X_1 and X_2 are two numbers on finite field $\mathbb{F} = \mathbb{Z} \bmod p$ where p is a large prime that $p > 2^N$. Without loss of generality, $X_1 < X_2$ if and only if $X_1 - X_2 \in [p - 2^{N-1}, p)$. Letting $X_3 = X_1 - X_2 - (p - 2^{N-1})$, we have $X_3 \in [0, 2^{N-1})$ and hence bit decomposition $\mathcal{C}_{bit}$ [44] can be used to check whether X_3 is in this range. The constraints for comparison $\mathcal{C}_{comp}(X_1 < X_2)$ can be written as follows:

$$\mathcal{C}_{comp}(X_1 < X_2) = \begin{cases} X_3 = X_1 - X_2 - (p - 2^{N-1}), \\ \mathcal{C}_{bit}(X_3, N-1, b_{X_3}), \end{cases} \tag{9}$$

where $b_{X_3} = [b_0, b_1, \ldots, b_{N-1}]$ denotes the binary representation of X_3. $\mathcal{C}_{comp}(X_1 < X_2) = 0$ suggests $X_1 < X_2$ holds and all constraints included are satisfied.

To reduce the number of constraints and thus the overhead of ZKP, we optimize the proof of $\mathcal{C}_{bit}(\cdot)$ employing the lookup table, a technique that trades space for time. Specifically, traditional bit decomposition needs to check the correctness of the binary form bit by bit, i.e., using $b_i \cdot (1 - b_i) = 0$ to check whether b_i equals 0 or 1, followed by checking whether the weighted sum of b_is equals X_3. To avoid the exorbitant overhead of bit checking, we propose to check X_3 by groups of 8 bits, i.e., verifying whether each group lies within the range of $[0, 255]$, and subsequently check if the weighted sum of all groups equals X_3. For each group, we predefine the range of $[0, 255]$ using the lookup table to reduce the number of constraints.

Summation. We take KGW as an example to illustrate the arithmetization method of summation. To count the number of green tokens $|y|_G$, we introduce an auxiliary flag variable $flg_i \in \{0, 1\}$ to indicate whether token y_i belongs to the green list G_i. The following constraint $\mathcal{C}_{flg}$ verifies the assignment of flg_i:

$$\mathcal{C}_{flg}(flg_i, r_i, \gamma|\mathbb{F}|) = \begin{cases} flg_i \cdot (1 - flg_i) = 0, \\ flg_i \cdot \mathcal{C}_{comp}(r_i < \gamma|\mathbb{F}|) = 0, \\ (1 - flg_i) \cdot \mathcal{C}_{comp}(\gamma|\mathbb{F}| < r_i) = 0, \end{cases} \tag{10}$$

where $\gamma|\mathbb{F}|$ denotes the pre-selected threshold and r_i represents the random number assigned to token y_i, which is computed by the aforementioned hash function and verified by $\mathcal{C}_{hash}$. Then we introduce a summation selector s_{Sum} and rewrite Eq. (1) as

$$s_{\text{Sum}} \cdot (x_1 + x_2 + \ldots + x_n) + s_O \cdot x_O = 0, \tag{11}$$

to construct the summation constraint $\mathcal{C}_{sum}(\text{Output}; \text{Inputs})$.

Algorithm 1 Proof Construction for KGW / SynthID-Text

Input: Secret key sk, the commitment of secret key H_{sk}, watermarked text $y = [y_0, \ldots, y_{n-1}]$, recent context window width ψ, green list size γ, threshold $\gamma \cdot |\mathbb{F}|$ / $\frac{|\mathbb{F}|}{2}$, the count result $|y|_G$ / S_g, the number of g-values assigned to each token ξ.

Output: Proof Π.

1: Build $\mathcal{C}_{hash}(H_{sk}; sk)$.
2: **for** $i = \psi, \psi + 1, \ldots, n - 1$ **do**
3: /* Arithmetize Step 1 */
4: Build $\mathcal{C}_{hash}(sd_i; sk, y_{\psi})$.
5: /* Arithmetize Step 2 */
6: Build $\mathcal{C}_{hash}(g^{y_i}; sd_i, y_i)$ / $\mathcal{C}_{hash}(g_k^{y_i}; sd_i, y_i, k)$ for $k \in \{1, \ldots, \xi\}$.
7: Set $flg^i = \mathbb{1}(g^{y_i} < \gamma \cdot |\mathbb{F}|)$ / $flg_k^i = \mathbb{1}(g_k^{y_i} < \frac{|\mathbb{F}|}{2})$ for $k \in \{1, \ldots, \xi\}$.
8: Build $\mathcal{C}_{flg}(flg^i, g^{y_i}, \gamma \cdot |\mathbb{F}|)$ / $\mathcal{C}_{flg}(flg_k^i, g_k^{y_i}, \frac{|\mathbb{F}|}{2})$ for $k \in \{1, \ldots, \xi\}$.
9: **end for**
10: Build $\mathcal{C}_{sum}((|y|_G$ / S_g; $Flg)$ where $Flg = \{flg^i | i \in \{\psi, \ldots, n - 1\}\}$ / $\{flg_k^i | i \in \{\psi, \ldots, n - 1\}, k \in [\xi]\}$.
11: $\mathcal{D}$ invokes ZKP.Prove$(u, w, \mathcal{C})$ to generate proof π where $u = \{y, \psi, \gamma|\mathbb{F}|/\frac{|\mathbb{F}|}{2}, |y_G|/S_g, \xi\}$ are the public input, $w = \{sk\}$ denotes the private input and $\mathcal{C}$ denotes the set of all constraints.
12: Return $\Pi = \{u, \pi\}$.

7 Applications of PVMark

In this section, we present how *PVMark* applies to different watermarking schemes to gain public verifiability. The optimized version using recursive ZKP is also demonstrated with examples.

7.1 Verifying Zero-& Multi-Bit Watermarks

PVMark is applied right after the watermark detection phase by prover $\mathcal{D}$, requiring the detection results $|y|_G$, S_g, COUNT and other constants as inputs. The output of proof construction is a proof π provided to verifier $\mathcal{V}$ who either passes or rejects the proof. Since the proving and verification follow the conventional ZKP protocols, we mainly focus on the proof construction. Hence the proof construction for KGW and SynthID-Text is introduced in Alg. 1 whereas that for Segment-Watermark is described in Alg. 2.

We show the proof construction of KGW and SynthID-Text in Alg. 1 with distinctions underlined. The algorithm mainly includes building $\mathcal{C}_{hash}$, $\mathcal{C}_{flg}$, and $\mathcal{C}_{sum}$. Then $\mathcal{D}$ generates proof π for all the constraints and releases π and the public inputs to $\mathcal{V}$ to carry out proving and verification following ZKP protocols.

Key commitment. To prevent inconsistent key use in the watermark embedding and detection, there is an additional step of key commitment in the embedding phase: the embedding party commits to sk by hash function $H_{sk} = H(sk)$

and releases H_{sk}. This one-way, collision-resistant function ensures the original key sk to be private, while preventing a key other than sk from producing the same commitment. Correspondingly, in the detection, the hashing of sk is modeled as a contraint (Line 1 of Alg. 1 and Alg. 2) to verify the consistency of the secret key. Actually, committing to sk before detection suffices to guarantee consistency.

Combining hash functions. In the original KGW scheme, ψ is set to 1, which allows us to make further adaptations by combining two hash functions (Line 3 and 5 of Alg. 1) into a three-to-one hash, thereby saving ZKP costs. Specifically, in round i, the random number g^{y_i} corresponding to the current token y_i is computed using two hash functions with two arguments: one calculates a random seed sd_i from sk and the previous token's index y_{i-1}, and the other uses sd_i and the current candidate token's index y_i to compute g^{y_i} (i.e., Step 1 and 2 in Table 2). We propose to replace them with a single function with three arguments:

$$g^{y_i} = \text{Hash}(sk, y_{i-1}, y_i),$$

which maps the tuple of the secret key, previous token index, and current candidate token's index to a single hash value. Therefore, for KGW, Lines 3 and 5 in Alg. 1 can be merged into 'Build $\mathcal{C}_{hash}(g^{y_i}; sk, y_{i-1}, y_i)$.' This optimization effectively reduces the circuit size and the ZKP costs in the detection process.

Note that in the original SynthID-Text, ψ is set to be greater than 1, e.g., $\psi = 4$. When using three-to-one hash functions to construct the constraint for Line 3 of Alg. 1, we combine pairs of hash functions into one. Therefore, $\mathcal{C}_{hash}(sd_i; sk, y_\psi)$ is defined as the set

$$\left\{ \begin{array}{l} \mathcal{C}_{hash}(H_1^{(i)}; sk, y_\psi^{(i)}[0], y_\psi^{(i)}[1]), \\ \mathcal{C}_{hash}(H_2^{(i)}; H_1^{(i)}, y_\psi^{(i)}[2], y_\psi^{(i)}[3]), \\ \qquad \vdots \\ \mathcal{C}_{hash}(sd_i; H_{\frac{\psi}{2}-1}^{(i)}, y_\psi^{(i)}[\psi-2], y_\psi^{(i)}[\psi-1]) \end{array} \right\}$$

where $H^{(i)}$ denotes the corresponding hash results.

We also present in Alg. 2 the proof construction of Segment-Watermark, which has an additional contraint $\mathcal{C}_{mem}$ for checking the token-to-position mapping, compared with Alg. 1.

7.2 Optimization with Recursive ZKP

We observe that for a chosen watermarking scheme and a selected hash function, the process of detecting the watermark for any token in text $y = [y_0, \ldots, y_{n-1}]$ follows the same steps, as shown in Line 2 to 7 of Alg. 1 and Line 2 to 8 of Alg. 2. This observation naturally leads us to construct the detection process by incrementally verifiable computation (IVC), as shown in Eq. (2). Take the KGW as an example, we reformulate the calculation of the sum of green tokens as

$$|y|_G^{(i+1)} = F(|y|_G^{(i)}, y_i). \tag{12}$$

Algorithm 2 Proof Construction for Segment-Watermark

Input: Secret key sk, the commitment of secret key H_{sk}, watermarked text $y = [y_0, \ldots, y_{n-1}]$, recent context window width ψ, green list size γ, threshold $\gamma \cdot |\mathbb{F}|$, token-to-position mapping $M : [0, |V|-1] \to [0, \hat{n}-1]$, the root value ROOT of the merkle tree corresponding to M, the count result matrix COUNT $\in \mathbb{Z}^{\hat{n} \times 2^{\hat{m}}}$.

Output: Proof Π.

1: Build $\mathcal{C}_{hash}(H_{sk}; sk)$.
2: **for** $i = \psi, \psi+1, \ldots, n-1$ **do**
3: Build $\mathcal{C}_{mem}(y_\psi, p_i, \text{ROOT})$.
4: **for** $j = 0, 1, \ldots, 2^{\hat{m}-1}$ **do**
5: Build $\mathcal{C}_{hash}(sd_i^j; sk, y_\psi, j)$.
6: Build $\mathcal{C}_{hash}(g_j^{y_i}; sd_i^j, y_i)$.
7: Set $flg_j^i = \mathbb{1}(g_j^{y_i} < \gamma \cdot |\mathbb{F}|)$.
8: Build a constraint using an addition gate for COUNT$[p_i][j] += flg_j^i$.
9: **end for**
10: **end for**
11: $\mathcal{D}$ invokes ZKP.Prove$(u, w, \mathcal{C})$ to generate proof π where $u = \{y, \psi, \gamma|\mathbb{F}|, \text{ROOT}, \text{COUNT}\}$ are the public input, $w = \{sk, M\}$ denotes the private input and $\mathcal{C}$ denotes the set of all constraints.
12: Return $\Pi = \{u, \pi\}$.

Within F, we compute the random number corresponding to y_i and compare it with the pre-selected threshold $\gamma|\mathbb{F}|$. If the random number is less than the threshold, y_i is a green token, and we set $|y|_G^{(i+1)} = |y|_G^{(i)} + 1$. After $n-1$ iterations, $|y|_G^{(n)}$ represents the count of green tokens in text y.

Given the incremental calculation, we can efficiently generate zero-knowledge proofs using Nova's folding scheme, turning the detection of each token in y as a relaxed R1CS instance. These instances are fold into one final instance by Eq. (4), (5) and (6). Finally, it suffices to use Nova to prove the correctness of the final instance and each folding process, which are represented as relaxed R1CS instances. However, a challenge arises as Nova requires the incremental value $|y|_G^{(i)}$ to be public, which does not meet the watermarking requirement that anything about the green token list should be private to prevent removal or ambiguity attacks.

To address this issue, we propose to disclose the hash of $|y|_G^{(i)}$ instead of the value. This modifies Eq. (12) to

$$\text{Hash}(|y|_G^{(i+1)}, s_H) = F(\text{Hash}(|y|_G^{(i)}, s_H), y_i), \tag{13}$$

where s_H is a private input introduced to prevent brute-force attacks on the hash. Since hashing is added, each instance adds two consistency check to verify 1) whether the private input $|y|_G^{(i)}$ of this round agrees with the output from the previous round, i.e., Line 3 of Alg. 3; 2) whether the public hash value $\text{Hash}(|y|_G^{(i+1)}, s_H)$ truely hashes the output of this round ($|y|_G^{(i+1)}$), i.e., Line 8 of Alg. 3.

Algorithm 3 Proof Construction w/ Recursive ZKP for KGW

Input: Same inputs with Alg. 1, number of tokens per instance N_t, number of foldings N_f and the secret s_H.
Output: proof Π.

1: Build $\mathcal{C}_{hash}(H_{sk}; sk)$.
2: Initialize the number of green tokens $|y|_G^{(1)} = 0$ and $H^{(1)} = \text{Hash}(0, s_H)$.
3: **for** $i = 1, 2, \ldots, N_f$ **do**
4: Build $\mathcal{C}_{hash}(|y|_G^{(i)}; s_H, H^{(i)})$ for consistency check.
5: **for** $j = 1, 2, \ldots, N_t$ **do**
6: Invoke Lines 2 to 7 in Alg. 1 to build the constraints $\mathcal{C}_{hash}$ and $\mathcal{C}_{flg}$.
7: **end for**
8: Accumulate the increment of $|y|_G^{(i)}$ to $|y|_G^{(i+1)}$ by adding up flg^j for $1 \le j \le N_t$ and establish the addition constraints.
9: Build $\mathcal{C}_{hash}(|y|_G^{(i+1)}; s_H, H^{(i+1)})$ for consistency check where $H^{(i+1)} = \text{Hash}(|y|_G^{(i+1)}, s_H)$.
10: Collect all the above constraints, represent them using relaxed R1CS, and derive a relaxed R1CS instance $Ins_{(i)}$.
11: **if** $i \neq 1$ **then**
12: Fold $Ins_{(i)}$ with $Ins'_{(i-1)}$ by establishing the constraints for folding process as in Eq. (4), (5) and (6) to obtain a folded instance $Ins'_{(i)}$ ($Ins'_{(1)} = Ins_{(1)}$).
13: **end if**
14: **end for**
15: Obtain the final instance $Ins'_{(N_f)}$.
16: Build $\mathcal{C}_{hash}(|y|_G^{(N_f+1)}; s_H, H^{(N_f+1)})$ and obtain an additional instance Ins_{ad}, where $|y|_G^{(N_f+1)}$ and $H^{(N_f+1)}$ are public inputs, and s_H is private input.
17: $\mathcal{D}$ invokes ZKP.Prove to generate proof π_1 and π_2 for $Ins'_{(N_f)}$ and Ins_{ad}, respectively.
18: Return $\Pi = \{\pi_1, \pi_2\}$.

Obtaining $\text{Hash}(|y|_G^{(n)}, s_H)$ through $n-1$ iterations of folding, we construct an additional instance on the hash value $\text{Hash}(|y|_G^{(n)}, s_H)$ and the public input $|y|_G^{(n)}$, with s_H being the private input, to prove the consistency between $\text{Hash}(|y|_G^{(n)}, s_H)$ and the public $|y|_G^{(n)}$, shown in Line 15 of Alg. 3. The value of $|y|_G^{(n)}$ is used for subsequent watermark score calculation.

In practical applications, there is always a tradeoff between the size of each instance and the number of instances, which leaves design choices to make. Overall, for the same computation, the larger the instance size, the higher the cost per instance but there would be fewer instances to verify. In *PVMark*, the choice is the number of tokens to be verified per instance. Intuitively, the folding scheme can save ZKP cost if the cost of proving an instance exceeds the cost of proving a folding. But the savings differ on the choices. Hence we analyze the tradeoff per case, as shown in Sec. 8.5.

We sum up our recursive ZKP for the detection of KGW in Alg. 3. The recursive ZKP for the detection of SynthID-Text is similar and presented in Alg. 4 of Appendix A.

Table 3: All variants of the implemented watermark detection algorithm, where ①②③④ denotes Groth16, PlonK, halo2 and Nova, respectively.

Scheme	Hash variants	MiMC	Poseidon	Poseidon2
KGW	Two-to-one	①②③④	①②③④	①②④
	Three-to-one	①②③	①②③	/
SynthID-Text	Two-to-one	①②④	①②④	①②④
Segment-Watermark	Two-to-one	④	④	④

Note that in the multi-bit watermarking scheme, the detection processes for different positions are not IVC processes, since the final value of the count result at one position is not equal to the initial value of the count result of the next position. In contrast, the detection process for the same position follows the paradigm of IVC. Therefore, we require the prover to group all tokens in the given text y according to their corresponding positions before generating the proof, and then use recursive ZKP to generate correctness proofs for the detection processes of each position in parallel. The recursive ZKP for the detection of Segment-Watermark is presented in Alg. 5 of Appendix A.

7.3 Scalability and Security Analysis

Scalability. *PVMark* is scalable, as the watermark detection process involves only random number generation, comparison, and score counting, all independent of vocabulary size or the underlying LLM. The ZKP costs are LLM-independent, ensuring efficient detection regardless of model size or vocabulary. *PVMark* generally scales with the folded size of token sequence length, except for the multi-bit instantiation which has a separate payload inherited from the underlying watermark scheme: for a message position with $\hat{m}$ bits, the detector exhausts a candidate space of size $2^{\hat{m}}$ to extract the bit string. Hence the ZKP costs grow with the candidate space to ensure the correctness and uniqueness of the decoded string. This is attributed to the inherent drawback of the multi-bit watermarking scheme, and can be addressed by a more efficient detection method.

Security analysis. Alg. 1 to Alg. 5 generate correctness proofs for watermark detection without exposing sk, and the probability of using an incorrect sk^* to deceive the verifier is negligible. This is guaranteed by the completeness, soundness, and zero-knowledge properties of zero-knowledge arguments, as proven in previous work [12, 17, 25]. Security proofs for Alg. 1 are provided in Appendix C, with similar proofs for the other algorithms.

8 Evaluations

Our adaption of the original watermarking schemes is based on the assumption that the hash function returns sufficiently random and uniformly distributed random values. Hence, for

Table 4: The avalanche effect coefficient and the chi-square statistics of hash functions. The latter three are qualified.

	SHA256 hash	Keccak256 hash	BLAKE2 hash	MiMC hash	Poseidon hash	Poseidon2 hash
Avalanche Effect Coeff.	0.498968	0.498948	0.498960	0.499570	0.499604	0.497579
Percent Rate (%)	0.206	0.210	0.208	0.086	0.079	0.484
$\chi^2 \pm s$	370.322 ± 40.496	372.057 ± 40.068	370.755 ± 40.157	10.309 ± 5.321	10.218 ± 5.180	10.209 ± 5.175
Success Rate (%)	0	0	0	99.1	99.2	99.4

Table 5: Optimal N_t of *PVMark* with Nova version v.s. different numbers of tokens (in the format of MiMC/Poseidon/Poseidon2 variants).

Scheme	**Number of tokens for watermark detection and verification**									
	200	400	600	800	1000	1200	1400	1600	1800	2000
KGW	25/40/20	40/40/25	40/50/30	40/50/40	50/100/50	60/75/48	70/70/50	50/80/50	72/90/60	80/100/80
SynthID-Text	8/8/5	10/8/5	12/15/8	16/16/10	20/20/10	20/20/12	20/20/14	20/25/16	25/25/18	25/25/20

PVMark to work in practice, we need to validate the assumption first. Besides, all properties of watermarking should be examined. Therefore, we aim to answer the following research questions by experiments on *PVMark*: 1) Does our adaptation using hash functions as PRF satisfy the randomness and uniformity assumption? 2) Does *PVMark* compromise the effectiveness, fidelity, and robustness of the original schemes? 3) How efficient is *PVMark*?

8.1 Implementation Details and Setup

The main part of the watermark embedding algorithm, which includes warping the logit distribution from the language model, is implemented using the `PyTorch` backend of the Huggingface library [52]. The hash function is implemented by `PyO3`, a Rust binding module for Python. In Rust, we implement SHA256, Keccak256, BLAKE2 for uniformity test and two-to-one and three-to-one variants of MiMC, Poseidon, and Poseidon2 as random number generators. To further accelerate execution, we utilize the Rust parallel computing library, `rayon`, to implement multi-threaded hash generation and comparison. In addition, by caching and reusing the green-list membership table induced by the fixed watermarking parameters, our implementation further improves the efficiency of generation and detection by replacing repeated membership computation with table lookup, without changing the watermarking rule.

For the proof construction of watermark detection, we use circom [20], a domain-specific language for defining arithmetic circuits, to construct constraints based on `circomlib` [21]. These constraints can be compiled into R1CS and PLONKish circuits through compiler module in circom. ZKP protocol Groth16 is used to verify R1CS whereas PLONK is used to verify PLONKish. We also implemented a variant of Plonk — halo2 [57], the KZG-based version, which supports custom gates and lookup operations thereby having a smaller circuit and better efficiency. We implement MiMC and Poseidon hash chips for halo2 to build PLONKish circuits. For the implementation of recursive ZKP, we use `Nova-Scotia` [36], which can compile circom circuits to Nova [33]. All variants of the implemented watermark detection are shown in Table 3.

Parameter settings. Our parameter choices follow three principles. First, for watermark embedding and detection, we follow the settings of the original work [7, 23, 39]. For KGW, we set the green list size $\gamma = 0.25$ and the context window width $\psi = 1$. For SynthID-Text, we set the context window width $\psi = 4$ and the number of g-values assigned to each candidate token $\xi = 30$. For Segment-Watermark, we set the green list proportion $\gamma = 0.5$, the context window width $\psi = 1$, the total number of positions $\hat{n} = 6$ in msg and the bit length $\hat{m} = 4$ per position. Second, for all ZKP protocols, we choose BN254, a pairing-friendly elliptic curve defined over a prime field that currently provides approximately 100-bit security level [22], widely used in blockchain project such as Ethereum [53] and Algorand [6]. BN254 offers faster computation due to its smaller field size compared to curves like BLS12–381. Third, for Nova, we select the per-instance token count N_t by minimizing the sum of the setup and the proving time for each token budget, rather than fixing N_t globally.

Models. We examine the performance of *PVMark* using the OPT-1.3B model [58], T5-Large model [41], Bloom-1.1B model [54], GPT2 [40] and Gemma-2B-IT [46]. A larger language model OPT-2.7B is used to compute perplexity (PPL) of texts.

Datasets and prompts. We select C4 dataset [41], Pile dataset [13] and ELI5 dataset [11] to evaluate the effectiveness, fidelity and robustness of *PVMark*. Specifically, we randomly select 100 texts with a length of more than 300 tokens from the news-like subset of the C4 dataset and 845 texts from all subsets of the Pile dataset, with each subset containing 20-50 texts. For each text, we remove the last 200 tokens and use the remaining portion as prompts for evaluation. Besides, we randomly select 1000 questions from ELI5's test set as prompts for evaluation.

Benchmark. We choose the original implementation of KGW [23] / SynthID-Text [7] to benchmark the watermarking

Table 6: Effectiveness & fidelity: success rate / z-score (mean score of g-values for SynthID-Text) / PPL of KGW and SynthID-Text adjusted for PVMark.

Watermarking Scheme	Model	Dataset	Benchmark	MiMC		Poseidon		Poseidon2
				two-to-one	three-to-one	two-to-one	three-to-one	two-to-one
KGW	OPT-1.3B	C4	99% / 9.78 / 11.84	99% / 9.86 / 12.88	97% / 9.75 / 13.99	96% / 10.86 / 11.22	96% / 10.45 / 13.97	99% / 9.40 / 12.97
		Pile	87% / 10.14 / 9.46	88% / 10.03 / 10.10	94% / 11.66 / 9.87	87% / 10.32 / 9.63	94% / 11.59 / 10.41	86% / 9.79 / 23.62
	T5-Large	C4	83% / 6.51 / 48.98	75% / 5.86 / 49.33	81% / 6.09 / 51.02	89% / 6.28 / 52.24	77% / 6.35 / 53.41	85% / 6.36 / 45.11
	Bloom-1.1B	C4	92% / 12.05 / 14.33	95% / 11.12 / 27.45	91% / 10.19 / 25.00	87% / 10.32 / 30.00	94% / 11.59 / 15.92	95% / 11.42 / 26.77
SynthID-Text	GPT2	ELI5	100% / 0.631 / 19.54	100% / 0.631 / 19.37	/	100% / 0.631 / 20.08	/	100% / 0.632 / 20.05
	Gemma-2B-IT	ELI5	96% / 0.548 / 13.38	91% / 0.546 / 13.27	/	97% / 0.547 / 12.81	/	96% / 0.547 / 13.75

Table 7: Success rate / watermark score / PPL of texts modified by three attack methods. The top, middle and bottom sections show texts generated by OPT-1.3B on C4, GPT2 on ELI5, and Gemma-2B-IT on ELI5.

Watermarking Scheme	Attack	Benchmark	MiMC		Poseidon		Poseidon2
			two-to-one	three-to-one	two-to-one	three-to-one	two-to-one
KGW	Attack (a)	85.86% / 5.75 / 74.33	88.89% / 5.79 / 78.41	87.50% / 5.92 / 87.68	91.67% / 6.38 / 67.81	86.46% / 6.35 / 83.80	80.81% / 5.38 / 79.53
	Attack (b)	54.55% / 4.10 / 140.95	44.44% / 3.91 / 138.75	48.96% / 3.98 / 134.65	46.88% / 4.26 / 123.37	56.25% / 4.27 / 157.34	32.32% / 3.35 / 139.47
	Attack (c)	84.85% / 6.03 / 17.14	83.84% / 6.09 / 17.72	83.33% / 6.07 / 18.92	95.83% / 7.30 / 15.14	89.58% / 6.67 / 18.84	77.78% / 5.46 / 18.83
SynthID-Text	Attack (a)	90.98% / 0.5321 / 108.19	88.60% / 0.5309 / 110.63	/	90.09% / 0.5326 / 106.66	/	88.40% / 0.5315 / 109.31
	Attack (b)	35.37% / 0.5114 / 195.95	31.00% / 0.5104 / 192.18	/	32.13% / 0.5102 / 188.64	/	29.30% / 0.5094 / 194.53
	Attack (c)	96.89% / 0.5355 / 25.00	92.20% / 0.5336 / 23.05	/	95.10% / 0.5349 / 23.08	/	93.30% / 0.5344 / 23.25
SynthID-Text	Attack (a)	34.20% / 0.5092 / 141.32	37.00% / 0.5094 / 127.83	/	33.60% / 0.5081 / 139.57	/	35.50% / 0.5088 / 123.15
	Attack (b)	12.60% / 0.5031 / 208.81	11.20% / 0.5027 / 207.05	/	10.70% / 0.5024 / 204.21	/	12.90% / 0.5034 / 200.69
	Attack (c)	35.20% / 0.5103 / 18.51	30.20% / 0.5086 / 19.37	/	29.10% / 0.5077 / 18.78	/	32.60% / 0.5089 / 17.92

performance.

Baselines. To compare *PVMark* with publicly detectable watermarking schemes, we include UPV [29] and PDW [10]. UPV exposes a public neural detector while keeping the watermark generation neural network private, whereas PDW embeds a publicly verifiable signature with error-correcting code enabled for decoding the signature. These schemes use different detector outputs, UPV probabilities and PDW boolean verification results are not directly comparable to KGW z-scores or SynthID g-value scores; we therefore compare them by the detection success rate, output fidelity, robustness, and efficiency.

All experiments are conducted on a Ubuntu 20.04 server with Intel(R) Xeon(R) Gold 6240C CPU with 256GB RAM and an NVIDIA GeForce RTX 3090 GPU.

8.2 Randomness and Uniformity of Hashes

We first evaluate the randomness and uniformity of hash functions to validate the proposal of using hash functions as pseudorandom number generators and the rationality of using a pre-selected threshold to divide the green and red tokens, or assign g-values.

We assess the avalanche effect and uniformity using the avalanche effect coefficient and the chi-square test. The avalanche effect is a desirable property of cryptographic hash functions, where flipping a single input bit should change about half of the output bits [51]. A coefficient close to 0.5 indicates better adherence. We compute the avalanche effect coefficient as described in [42], using $N_1 = N_2 = 256$ for SHA256, Keccak256, and BLAKE2, and $N_1 = N_2 = 254$ for MiMC, Poseidon, and Poseidon2, averaging over 5,000,000 iterations. We show the closeness between the coefficient c and 0.5 by calculating the percentage of the difference, i.e., $\frac{|c-0.5|}{0.5}$. A smaller percentage indicates better randomness.

To verify the uniformity, we conduct a chi-square goodness of fit test, a statistical hypothesis test. The null hypothesis is 'all random numbers used for vocabulary partitioning in each round are uniformly distributed in the finite field $\mathbb{F}$ defined by BN254.' If the result χ^2 is less than the threshold, the null hypothesis is accepted, meaning that the random numbers used for vocabulary division are uniformly distributed within the value range. We average the chi-square test statistics χ^2 over $100,000$ iterations. In each iteration, we randomly select a secret key $sk \in \mathbb{F}$, a previous token index $y' \in [0, |V| = 50265)$, and the current token index $y'' = [0, |V|)$ to generate $|V|$ random numbers. The statistic threshold is 26.296 for significance level $\alpha = 0.05$.

The avalanche effect coefficient of hash functions and the percent rate of the coefficient v.s. 0.5 are shown in the top part of Table 4. The average and standard deviation of multiple chi-squared tests statistic χ^2, along with the success rate of passing the tests, are shown at the bottom of Table 4. It can be observed that the avalanche effect coefficients of the six types of hash functions are all close to 0.5, indicating sufficient randomness. On $\mathbb{F}$, MiMC, Poseidon, and Poseidon2 hash exhibit uniformity, thereby qualifying for our scheme.

Since we have validated the randomness and uniformity of hash functions used in *PVMark*, our adaption theoretically does not affect the original watermarking scheme, i.e., an almost equivalent performance should be achieved with *PVMark*.

Table 8: The embedding time (WET) and detection time (WDT) excluding ZKP parts. Each entry is A/B, where A is ms/token and B is s/sample for 200 tokens. KGW is used on OPT-1.3B and SynthID-Text is applied on GPT-2.

Watermarking Scheme	Metric	Benchmark	MiMC		Poseidon		Poseidon2
			two-to-one	three-to-one	two-to-one	three-to-one	two-to-one
KGW	WET	0.2640 / 0.0528	0.2356 / 0.0471	0.2369 / 0.0473	0.2401 / 0.0480	0.2350 / 0.0470	0.2347 / 0.0469
	WDT	0.1447 / 0.0289	0.0377 / 0.0075	0.0217 / 0.0043	0.0184 / 0.0037	0.0161 / 0.0032	0.0059 / 0.0012
SynthID-Text	WET	1.9970 / 0.3994	2.2226 / 0.4445	/	2.5168 / 0.5034	/	1.8509 / 0.3702
	WDT	0.1594 / 0.0319	0.0355 / 0.0071	/	0.0409 / 0.0082	/	0.0217 / 0.0043

Table 9: Effectiveness, robustness, and efficiency of publicly detectable baselines. Notation follows Tables 6, 7, and 8. For efficiency, UPV uses 200-token samples, while PDW uses full-signature variable-length generation.

Baseline	Model	Dataset	Effectiveness	Robustness (SR / PPL)			Efficiency	
			SR / PPL	Attack (a)	Attack (b)	Attack (c)	WET	WDT
UPV [29]	OPT-1.3B	C4	95.00% / 11.66	0.00% / 76.73	73.68% / 134.36	61.05% / 19.36	9.5393 / 1.9079	0.0093 / 0.0019
	GPT-2	ELI5	98.80% / 8.28	0.40% / 49.91	63.50% / 91.84	88.10% / 9.65	5.4175 / 1.0835	0.0104 / 0.0021
PDW [10]	OPT-1.3B	C4	91.00% / 30.84	0.00% / 143.55	0.00% / 215.57	0.00% / 40.95	117.68 / 23.54	0.0184 / 0.0037
	GPT-2	ELI5	98.50% / 188.41	0.00% / 483.02	0.00% / 500.35	0.00% / 165.51	30.50 / 6.10	2.2472 / 0.4494

Table 10: Reverse-training attack on UPV under GPT-2 on ELI5. Cracking metrics measure generator-rule recovery, while detected/forged and plain FP report detector-level acceptance rates of the forged and plain texts.

Window size	Cracking F1	Cracking Acc.	Detected/Forged	Plain FP
1	0.991	99.14%	100/100	43/100
2	0.796	72.57%	96/100	74/100
3	0.698	75.37%	100/100	28/100
5	0.667	68.76%	44/100	3/100

8.3 Effectiveness and Fidelity

We evaluate the effectiveness and fidelity of KGW and SynthID-Text adjusted for PVMark on real datasets. Three metrics are used to evaluate the effectiveness: success rate (SR), i.e., the rate of successfully detected watermarks; watermark score, i.e., z-score / mean score of g-values, which indicates the watermark strength (z-score greater than 4.0 or mean score of g-values greater than 0.514 is considered strong indicative of watermark presence); and perplexity (PPL), which reflects text quality—the lower the value, the higher the text quality. The results are shown in Table 6.

It can be observed that when using the OPT-1.3B, Bloom-1.1B, GPT2 and Gemma-2B-IT models, various variants of KGW and SynthID-Text adjusted for PVMark demonstrate comparable effectiveness and fidelity with the benchmark, indicating strong watermarking performance. The performance on Pile is inferior to that on C4, OPT-1.3B, since Pile contains code which has different syntatic rules from natural language. The performance of T5-large model is consistently lower across all, mostly due to its weaker generation performance.

Table 9 reports the effectiveness of publicly detectable baselines. UPV achieves high SRs of 95.00% and 98.80%, while PDW achieves 91.00% and 98.50% on the two evaluated settings. Compared with these baselines, *PVMark* mostly preserves the effectiveness of the underlying KGW and SynthID-Text schemes (Table 6), reaching up to 99% SR for KGW and 100% SR for SynthID-Text with comparable PPL to the original schemes.

8.4 Robustness

We investigate the robustness of KGW and SynthID-Text adjusted for PVMark against three removal attacks: a) random word deletion, b) random synonym substitution with WordNet [34], and c) context-aware synonym substitution with BERT [8] as the embedding model. We reuse the implementation of MarkLLM [47], setting the deletion ratio for a) to 0.3 and the substitution ratio to 0.5 for b) and c). The success rate (SR), i.e., the rate of the successfully detected watermark, watermark score and PPL of the modified text are recorded.

By Table 7, it is evident that the average watermark score of the attacked text decreases, indicating a reduction in watermark strength. *PVMark* with MiMC, Poseidon and Poseidon2 variants shows performance comparable to the benchmark in resisting attacks. The high PPL, potentially associated with the generation capability of the test model, does not affect our robustness testing.

Table 9 shows that publicly detectable baselines have different robustness performance. UPV is brittle under random word deletion, dropping to 0.00% and 0.40% SRs on the two evaluated settings, although it retains moderate robustness under synonym-based substitutions. This is because deletion attack would shift the local window used to detect the watermark. PDW is more fragile under the three edit attacks, with post-attack SR dropping to 0.00% for random deletion, WordNet synonym substitution, and context-aware synonym substitution, since PDW heavily relies on the exact 'consecutive tokens' to verify the signature and hinges upon the error-correcting code for robust decoding. In comparison, *PVMark* inherits the robustness profile of KGW and SynthID-Text,

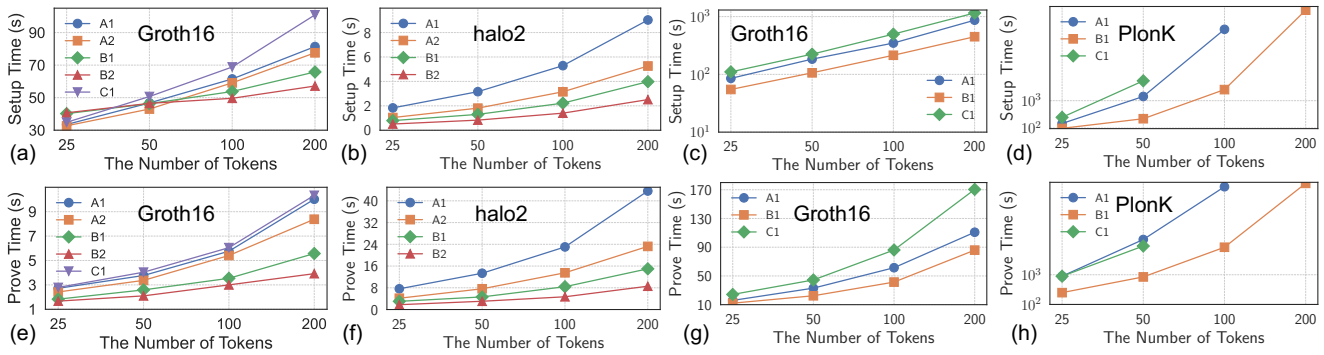

Figure 2: ZKP costs for *PVMark* v.s. different numbers of tokens: the left two columns are for detection by KGW and the right two columns are for detection by SynthID-Text. A, B, C denote MiMC, Poseidon, Poseidon2 variants, respectively and 1 and 2 represent the use of two-to-one hash and three-to-one hash.

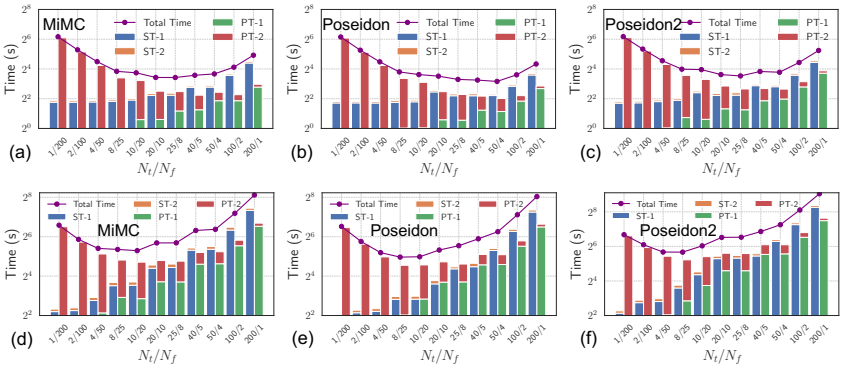

Figure 3: ZKP costs for Nova version of *PVMark* v.s. varying N_t, N_f ($N_t \times N_f = 200$ tokens): the upper row is for detection by KGW and the lower row is for detection by SynthID-Text. ST-1, ST-2 denote the setup time to prove the final instance and the folding process, respectively, and PT-1, PT-2 are their prove time correspondingly. The total time cost is ST-1 + ST-2 + PT-1 + PT-2.

Table 11: Optimal N_t of *PVMark* with Nova version v.s. different numbers of tokens (in the format of MiMC/Poseidon/Poseidon2 variants).

Scheme	Number of tokens					
	60	120	240	480	960	1920
Segment	1/1/1	2/2/2	2/2/2	4/4/2	5/5/1	5/8/5

Table 12: Proof sizes (KB) of the detection of KGW for 200 tokens. The most cost-effective N_t, N_f are adopted for Nova.

ZKP	MiMC		Poseidon		Pos2
	2to1	**3to1**	**2to1**	**3to1**	**2to1**
Groth16	0.8	0.8	0.8	0.8	0.8
halo2	14	14	14	14	/
PlonK	2.3	2.3	2.3	2.3	2.3
Nova	10.2	/	10.5	/	10.5

providing stabler robustness across the evaluated edit attacks in Table 9.

Table 10 further evaluates UPV under a reverse-training attack, which targets a different robustness dimension from text-edit attacks. Following the setting of [29], we let the attacker query the public UPV detector to collect 10,000 pairs of input-output, ranging from 100 to 200 tokens. The collected data is used to train a surrogate generator to guide the LLM to generate forged text. UPV is sensitive to the window size: for $w = 1$, the attacker nearly recovers the generator rule, with a cracking F1 of 0.991 and producing 100/100 forged text accepted by the detector. For the default $w = 5$ setting, the attack does not fully recover the generator rule, with a cracking F1 of merely 0.667; however, it still shows non-negligible detector-level forgeability, with 44/100 forged texts accepted compared with only 3/100 false positives on plain text. This highlights a design-level advantage of *PVMark*: it keeps the original detector and secret key private, and exposes only a ZKP certificate of correct detection. Therefore, the reverse-training attack against a public detector is not directly applicable to *PVMark*; public verifiability is obtained without exposing a learnable detection interface.

8.5 Efficiency

The efficiency of *PVMark* is evaluated by two separate parts. The first one is the watermarking associated cost, including watermark embedding time (WET) and watermark detection time (WDT) excluding the ZKP parts. The second part is the cryptographic cost required for public verifiability, including setup, proof generation, verification, and proof size.

Watermark embedding and detection costs. Table 8 reports the watermarking associated costs. As an engineering optimization, we use a pre-computed lookup table to store hash-based green-list membership in the KGW embedding. The lookup table takes about 2.35GB of storage and ~ 20 minutes to initialize offline. KGW embedding under the same watermark parameters can reuse the table to avoid repeated hash computation online. This optimization is not adopted for SynthID-Text.

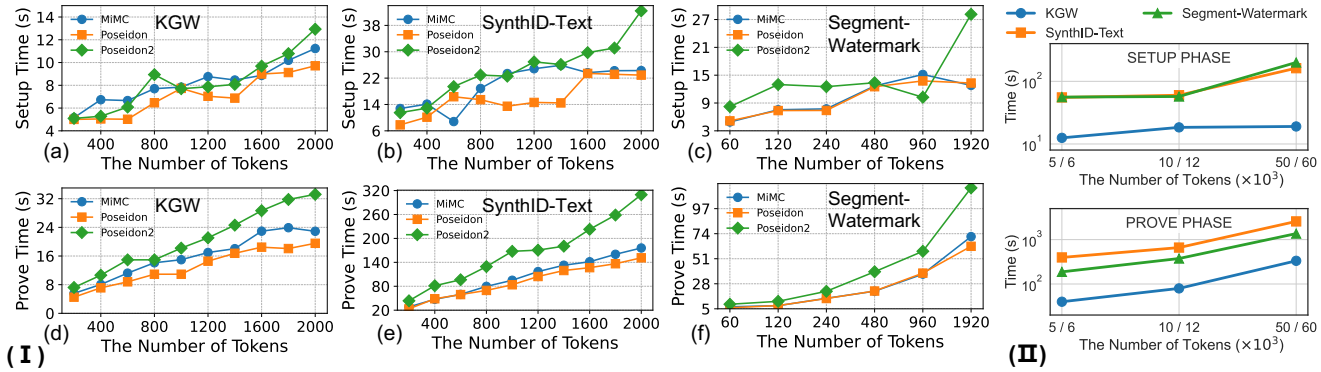

Figure 4: Evaluation of ZKP costs for the Nova version of *PVMark*. **(I)** Detailed comparison of different hash functions (MiMC, Poseidon, Poseidon2) across varying token counts for KGW, SynthID-Text, and Segment-Watermark. **(II)** Scalability analysis of the Poseidon-instantiated *PVMark* on very large token counts. In all experiments, the setting of N_t is optimized such that the sum of setup time and prove time is minimized. Note that in (II), the x-axis labels denote the number of tokens for KGW & SynthID-Text / Segment-Watermark, respectively.

Table 8 shows that our hash-based adaptation does not introduce a bottleneck in the online watermarking path. For KGW, *PVMark* is slightly faster than the benchmark in WET. For SynthID-Text, the WET of *PVMark* remains at the same level as the benchmark, and the Poseidon2 variant is even faster. The detection cost is even lower after adaptation. For both KGW and SynthID-Text, *PVMark* reduces WDT compared with the benchmark, with Poseidon2 achieving the lowest detection latency among the evaluated variants. This efficiency is mainly attributed to our engineering optimizations, including the Rust backend, parallelized hash computation, and effective cache reuse. These results indicate that replacing the original pseudorandom procedures with ZKP-friendly hash-based algorithms does not slow down normal generation.

Table 9 further compares the watermarking efficiency of publicly detectable baselines. UPV has low detection latency but incurs high embedding costs. The WET of PDW is measured in full-signature variable-length generation and thus is averaged over 200 tokens. The WDT of PDW reports the latency of successful detections; the latency of rejecting unwatermarked text is much longer. Compared with baselines, *PVMark*'s watermark embedding and detection overhead (shown in Table 8) remains small, making it lightweighted to deploy in the LLM generation pipeline.

Proof generation for zero-bit watermarking schemes. Fig. 2 and Fig. 3 present ZKP costs for *PVMark* variants using four ZKP protocols: Groth16, Halo2, Plonk, and Nova. Costs include setup time, prove time, verification time, and proof size. Verification time is not shown, as even the slowest (Plonk) verification takes under two seconds, much less than setup and prove time.

In Fig. 2, while Groth16 has a higher setup time than Halo2, its prove time is shorter. Overall, Halo2 is 1-5× faster due to optimized circuit design and lookup tables. Plonk, however, suffers from high setup and prove times due to its use of the universal Circom compiler and the circom programming language, preventing us from tailoring the circuit to the application. Among hash variants, the three-to-one hash reduces costs compared to the two-to-one hash, with savings increasing as more tokens are verified. Among all hash variants, Poseidon hash has the lowest overhead, making it ideal for ZKP protocols. Compared to KGW, the detection of SynthID-Text is more costly. This is attributed to the more complicated process of Tournament sampling and thereby larger ZKP circuits. In fact, the circuit is so large that the memory demand is too high for us to fully measure the overhead of PlonK.

Since the original settings of KGW and SynthID-Text mostly involve a large number of tokens in watermark detection, we thus focus on the case of verifying 200 or more tokens. In that case, recursive ZKP shows a much lower cost than others as in Fig. 3. It is observed that the setup time for the folding process is almost constant and negligible compared to other parts. The setup time and prove time for the final instance increase with a rise in N_t as a single instance grows larger. Meanwhile, the decrease in N_f leads to a reduction in the prove time for the folding process. We find that for the MiMC, Poseidon, and Poseidon2 variants of *PVMark*, setting N_t to 20, 50, and 25 accordingly results in the minimum total overhead, which are around $2^4, 2^5$s for KGW, SynthID-Text, respectively. The overhead of detecting a larger number of tokens is shown in Fig. 4. When detecting 2000 tokens, the least total time cost (setup and prove) is approximately 30s and 175s for KGW and SynthID-Text, respectively, where the optimal N_t is shown in Table 5. Hence we consider the running time is reasonable in a watermark verification setting. For detecting very long texts (*e.g.,* 50,000 tokens), the total computational overhead for the KGW scheme is approximately 352s, including setup and proof generation. This latency remains acceptable for practical watermark verification scenarios.

Table 12 shows the proof size required in detection. Groth16 has the smallest proof size, less than 1KB. Actually, the proof sizes of Groth16, halo2, and PlonK are constant for 25/50/100/200 tokens. This is because Groth16's proof requires only 3 pairing checks, involving 3 group elements, regardless of the number of constraints. Both halo2 and PlonK use KZG commitments, of which the proof contains a limited group element, independent of the number of constraints. Nova utilizes Spartan [43] at its core, a ZKP where proof size is logarithmically related to the number of constraints, but still the proof size is adequately small for 200 tokens.

Proof generation for multi-bit watermarking schemes. Compared to the zero-bit watermark, it takes more constraints to verify the detection of Segment-Watermark. Hence we choose the Nova variant of *PVMark* to generate proofs for cost consideration. It is worth noting that the detection for different positions of the message in Segment-Watermark can be performed in parallel. Thus we display in the rightmost of Fig. 4(I) the ZKP costs required for different numbers of tokens in verifying a single position of the watermark.

It can be observed that when the Poseidon hash function is used as the pseudorandom generator, the ZKP costs are minimized. When the number of tokens to be verified is 1920 (i.e., for 6 positions, with each position requiring checking 320 tokens), the optimal setup time and prove time are approximately 13 and 62 seconds, which we consider to be a fully acceptable cost. The optimal N_t is shown in Table 11. These experiments use the original Segment-Watermark payload setting $\hat{m} = 4$. We further evaluate larger payload settings, $\hat{m} = 5$ and $\hat{m} = 7$, in Appendix B, with the results shown in Table 13.

9 Conclusion

To resolve the trust issue in LLM watermark detection, we propose *PVMark*, a plugin that enables zero-bit and multi-bit LLM watermarking schemes to gain public verifiability without compromising the original schemes' performance. The design is featured by the application of zero-knowledge proof to prove the correctness of watermark detection process, without revealing any credential. To achieve this, we redesigned the watermark embedding and detection schemes to align with ZKP, including introducing hash functions as PRFs, customizing the circuits, applying recursive ZKP, etc. We hope *PVMark* not only contributes to the watermarking community, but also to a wider area of applied ZKP research.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their valuable feedback. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant Nos. 62272306, U25A20445, and 62432008, by the CCF-Ant Research Fund under Grant No. CCF-AFSG RF20240407, by RGC RIF Grant No. R6021-20, RGC TRS Grant No. T43-513/23N-2, RGC CRF Grant Nos. C7004-22G, C1029-22G, and C6015-23G, NSFC/RGC Grant No. CRS_HKUST601/24, and RGC GRF Grant Nos. 16207922, 16207423, and 16203824.

Ethical Considerations

Overview. PVMark enables public verification of LLM watermark detection through zero-knowledge proofs (ZKPs). It is designed for settings where a third party needs to verify a detection result, but revealing the watermark secret would weaken the watermarking scheme itself. PVMark therefore proves correct execution of the detection procedure without disclosing the secret key. Our analysis is guided by a stakeholder-based view of beneficence, respect for persons, justice, and public interest.

Stakeholders. PVMark may affect end users, content platforms, model providers, accused or suspected parties, researchers, legal and regulatory actors, and potential adversaries. These groups have different interests in provenance, accountability, contestability, system security, and responsible use.

Benefits. PVMark can make watermark-based provenance claims more credible by allowing independent verification without exposing long-lived secrets. This is useful in audits, disputes, and moderation workflows, where verification should not itself create new opportunities for watermark evasion. More broadly, PVMark shows how ZKPs can support public accountability while preserving confidentiality.

Risks and Mitigations. The main risks come from overuse or overinterpretation of watermark evidence. A PVMark proof shows only that a specified detection algorithm was correctly executed under a committed key on a given input. It does not establish authorship, intent, or exclusive access to a particular model. Watermark evidence may also be weakened by paraphrasing, translation, truncation, partial quotation, or adversarial editing. If treated as conclusive, such evidence could lead to over-enforcement, censorship, or unfair claims against individuals.

We mitigate these risks in several ways. The work does not involve human subjects, deception, or private user data, and experiments are conducted in controlled offline settings using open-source models and toolchains. At the system level, PVMark protects watermark secrets and separates detection claims from broader attribution claims. In deployment, PVMark outputs should be treated as one piece of evidence rather than a standalone basis for punitive action. Systems using PVMark should document their assumptions, report false positive and false negative behavior, and provide appeal or review mechanisms, especially in high-stakes settings.

Limitations. PVMark does not solve the robustness limits of watermarking schemes themselves, including vulnerability

to editing, paraphrasing, translation, and truncation. As with any ZKP-based system, mistakes in circuits, constraints, or implementations may cause the proof to capture a different statement than intended. Independent review of the proof circuits and verification logic is therefore important before high-stakes deployment.

Justice, Law, and Public Interest. The benefits of PVMark may accrue mainly to model providers and large platforms that can deploy cryptographic verification systems, while enforcement burdens may fall on individual users or smaller organizations. Fair use of PVMark requires clear communication of what is and is not proven, meaningful opportunities to contest decisions, and compliance with applicable norms and laws concerning privacy, surveillance, consumer protection, evidence, and procedural fairness.

Decision to Proceed and Publish. We believe the benefits of PVMark outweigh its foreseeable risks when its scope and limitations are clearly stated. PVMark is a defensive and transparency-oriented contribution: it does not introduce a watermark removal method, but instead offers a more accountable way to verify existing watermark detectors. Any released artifacts should avoid including secret keys or materials that would directly facilitate watermark circumvention.

Open Science

The source code is available at https://github.com/ashleyxly/PVMark and archived on Zenodo at https://doi.org/10.5281/zenodo.20406335.

References

[1] Key Rotation - Glossary | CSRC — csrc.nist.gov. https://csrc.nist.gov/glossary/term/key_rotation. [Accessed 05-02-2026].

[2] In re Micron Technology Inc. https://fedcircuitblog.com/wp-content/uploads/2025/06/25-117.ORDER_.2-26-2025_2473455.pdf, 2025.

[3] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, pages 191–219. Springer, 2016.

[4] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators, 2015-06-24 2015. doi:10.6028/NIST.SP.800-90Ar1.

[5] Lenore Blum, Manuel Blum, and Mike Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on computing*, 15(2):364–383, 1986.

[6] Jing Chen and Silvio Micali. Algorand, 2017. URL: https://arxiv.org/abs/1607.01341, arXiv:1607.01341.

[7] Sumanth Dathathri, Abigail See, Sumedh Ghaisas, Po-Sen Huang, Rob McAdam, Johannes Welbl, Vandana Bachani, Alex Kaskasoli, Robert Stanforth, Tatiana Matejovicova, et al. Scalable watermarking for identifying large language model outputs. *Nature*, 634(8035):818–823, 2024.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, pages 4171–4186, 2019.

[9] Ahmed M Elkhatat, Khaled Elsaid, and Saeed Almeer. Evaluating the efficacy of ai content detection tools in differentiating between human and ai-generated text. *International Journal for Educational Integrity*, 19(1):17, 2023.

[10] Jaiden Fairoze, Sanjam Garg, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, and Mingyuan Wang. Publicly-detectable watermarking for language models, 2024. URL: https://arxiv.org/abs/2310.18491, arXiv:2310.18491.

[11] Angela Fan, Yacine Jernite, Ethan Perez, David Grangier, Jason Weston, and Michael Auli. ELI5: Long form question answering. In *ACL*, pages 3558–3567, 2019.

[12] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge, 2019. URL: https://eprint.iacr.org/2019/953.

[13] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020. URL: https://arxiv.org/abs/2101.00027, arXiv:2101.00027.

[14] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security Symposium*, pages 519–535, 2021.

[15] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the poseidon hash function. In *AFRICACRYPT*, pages 177–203, 2023.

[16] Benjamin Greenberg and Jessica Leano. A cautionary tale of contempt proceedings for potential violation of a standard protective order in trade secrets litigation. https://www.thetmca.com/a-cautionary-tal

e-of-contempt-proceedings-for-potential-violation-of-a-standard-protective-order-in-trade-secrets-litigation/, 2024.

[17] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326, 2016.

[18] Abhimanyu Hans, Avi Schwarzschild, Valeriia Cherepanova, Hamid Kazemi, Aniruddha Saha, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Spotting llms with binoculars: zero-shot detection of machine-generated text. In *ICML*, 2024.

[19] Zhengmian Hu, Lichang Chen, Xidong Wu, Yihan Wu, Hongyang Zhang, and Heng Huang. Unbiased watermark for large language models. In *ICLR*, 2024.

[20] iden3. Circom. https://github.com/iden3/circom, 2021.

[21] iden3. circomlib. https://github.com/iden3/circomlib, 2021.

[22] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *CRYPTO*, pages 543–571, 2016.

[23] John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. A watermark for large language models. In *ICML*, pages 17061–17084, 2023.

[24] John Kirchenbauer, Jonas Geiping, Yuxin Wen, Manli Shu, Khalid Saifullah, Kezhi Kong, Kasun Fernando, Aniruddha Saha, Micah Goldblum, and Tom Goldstein. On the reliability of watermarks for large language models. In *ICLR*, 2024.

[25] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *CRYPTO*, pages 359–388, 2022.

[26] Kalpesh Krishna, Yixiao Song, Marzena Karpinska, John Wieting, and Mohit Iyyer. Paraphrasing evades detectors of ai-generated text, but retrieval is an effective defense. *NeurIPS*, 2024.

[27] Rohith Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. Robust distortion-free watermarks for language models. *TMLR*, 2024.

[28] 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 *ACL*, pages 4890–4911, 2024.

[29] Aiwei Liu, Leyi Pan, Xuming Hu, Shuang Li, Lijie Wen, Irwin King, and S Yu Philip. An unforgeable publicly verifiable watermark for large language models. In *ICLR*, 2024.

[30] Yepeng Liu and Yuheng Bu. Adaptive text watermark for large language models. In *ICML*, pages 30718–30737, 2024.

[31] George Marsaglia. Xorshift rngs. *Journal of Statistical software*, 8:1–6, 2003.

[32] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[33] microsoft. Nova. https://github.com/microsoft/Nova, 2022.

[34] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[35] Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In *ICML*, pages 24950–24962, 2023.

[36] nalinbhardwaj. Nova-scotia. https://github.com/nalinbhardwaj/Nova-Scotia, 2023.

[37] Yikang Pan, Liangming Pan, Wenhu Chen, Preslav Nakov, Min-Yen Kan, and William Wang. On the risk of misinformation pollution with large language models. In *Findings of EMNLP*, pages 1389–1403, 2023.

[38] Qi Pang, Shengyuan Hu, Wenting Zheng, and Virginia Smith. No free lunch in llm watermarking: Trade-offs in watermarking design choices. *NeurIPS*, 37:138756–138788, 2024.

[39] Wenjie Qu, Wengrui Zheng, Tianyang Tao, Dong Yin, Yanze Jiang, Zhihua Tian, Wei Zou, Jinyuan Jia, and Jiaheng Zhang. Provably robust multi-bit watermarking for {AI-generated} text. In *USENIX Security Symposium*, pages 201–220, 2025.

[40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[42] Arish Sateesan. Analyze your hash functions: The avalanche metrics calculation, 2020. URL: https://arishs.medium.com/analyze-your-hash-functions-the-avalanche\-metrics-calculation-767b7445ee6f.

[43] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *CRYPTO*, pages 704–737, 2020.

[44] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*, pages 253–268, 2012.

[45] Robert C Tausworthe. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19(90):201–209, 1965.

[46] Gemma Team. Gemma: Open models based on gemini research and technology, 2024. URL: https://arxiv.org/abs/2403.08295, arXiv:2403.08295.

[47] THU-BPM. Markllm. https://github.com/THU-BPM/MarkLLM, 2024.

[48] Christoforos Vasilatos, Manaar Alam, Talal Rahwan, Yasir Zaki, and Michail Maniatakos. Howkgpt: Investigating the detection of chatgpt-generated university student homework through context-aware perplexity analysis, 2023. URL: https://arxiv.org/abs/2305.18226, arXiv:2305.18226.

[49] Vivek Verma, Eve Fleisig, Nicholas Tomlin, and Dan Klein. Ghostbuster: Detecting text ghostwritten by large language models. In *NAACL*, pages 1702–1717, 2024.

[50] Lean Wang, Wenkai Yang, Deli Chen, Hao Zhou, Yankai Lin, Fandong Meng, Jie Zhou, and Xu Sun. Towards codable watermarking for injecting multi-bits information to LLMs. In *ICLR*, 2024.

[51] Arthur F Webster and Stafford E Tavares. On the design of s-boxes. In *EUROCRYPT*, pages 523–534, 1985.

[52] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, et al. Transformers: State-of-the-art natural language processing. In *EMNLP*, pages 38–45, 2020.

[53] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[54] BigScience Workshop. Bloom: A 176b-parameter open-access multilingual language model, 2023. URL: https://arxiv.org/abs/2211.05100, arXiv:2211.05100.

[55] Yihan Wu, Zhengmian Hu, Junfeng Guo, Hongyang Zhang, and Heng Huang. A resilient and accessible distribution-preserving watermark for large language models. In *ICML*, 2024.

[56] KiYoon Yoo, Wonhyuk Ahn, Jiho Jang, and Nojun Kwak. Robust multi-bit natural language watermarking through invariant features. In *ACL*, pages 2092–2115, 2023.

[57] Zcash. halo2. https://github.com/zcash/halo2, 2022.

[58] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, et al. Opt: Open pre-trained transformer language models, 2022. URL: https://arxiv.org/abs/2205.01068, arXiv:2205.01068.

[59] Xuandong Zhao, Prabhanjan Vijendra Ananth, Lei Li, and Yu-Xiang Wang. Provable robust watermarking for AI-generated text. In *ICLR*, 2024.

[60] Xuandong Zhao, Sam Gunn, Miranda Christ, Jaiden Fairoze, Andrés Fábrega, Nicholas Carlini, Sanjam Garg, Sanghyun Hong, Milad Nasr, Florian Tramèr, Somesh Jha, Lei Li, Yu-Xiang Wang, and Dawn Song. Sok: Watermarking for ai-generated content. In *S&P*, pages 2621–2639, 2025.

Algorithm 4 Proof Construction Using Recursive ZKP for SynthID-Text

Input: Same inputs with Alg. 1, number of tokens per instance N_t, number of foldings N_f and the secret s_H.
Output: proof Π.

1: Build $\mathcal{C}_{hash}(H_{sk}; sk)$.
2: Initialize the total sum of g-values $S_g^{(1)} = 0$.
3: Generate the context token set $y_\psi^{(i)}$ for y_i $(\psi \le i \le n-1)$.
4: **for** $i = 1, 2, \ldots, N_f$ **do**
5: Build $\mathcal{C}_{hash}(S_g^{(i)}, s_H, H^{(i)})$ for consistency check.
6: **for** $j = 1, 2, \ldots, N_t$ **do**
7: Invoke Lines 2 to 7 in Alg. 1 and use the corresponding y_ψ as one of the inputs to build the constraints $\mathcal{C}_{hash}$ and $\mathcal{C}_{flg}$.
8: **end for**
9: Accumulate the increment of $S_g^{(i)}$ to $S_g^{(i+1)}$ by adding up flg_k^j for $j \in [N_t], k \in [\xi]$ and establish the addition constraints.
10: Build $\mathcal{C}_{hash}(S_g^{(i+1)}, s_H, H^{(i+1)})$ for consistency check where $H^{(i+1)} = \text{Hash}(S_g^{(i+1)}, s_H)$.
11: Collect all the above constraints, represent them using relaxed R1CS, and derive a relaxed R1CS instance $Ins_{(i)}$.
12: **if** $i \neq 1$ **then**
13: Fold $Ins_{(i)}$ with $Ins'_{(i-1)}$ by establishing the constraints for folding process as in Eq. (4), (5) and (6) to obtain a folded instance $Ins'_{(i)}$ $(Ins'_{(1)} = Ins_{(1)})$.
14: **end if**
15: **end for**
16: Obtain the final instance $Ins'_{(N_f)}$.
17: Build $\mathcal{C}_{hash}(S_g^{(N_f+1)}, s_H, H^{(N_f+1)})$ and obtain an additional instance Ins_{ad}, where $S_g^{(N_f+1)}$ and $H^{(N_f+1)}$ are public inputs, and s_H is private input.
18: $\mathcal{D}$ invokes ZKP.Prove to generate proof π_1 and π_2 for $Ins'_{(N_f)}$ and Ins_{ad}, respectively.
19: Return $\Pi = \{\pi_1, \pi_2\}$.

Algorithm 5 Proof Construction Using Recursive ZKP for Segment-Watermark

Input: Same inputs with Alg. 2, number of tokens per instance N_t, number of foldings N_f and the secret s_H.
Output: proof Π.

1: Build $\mathcal{C}_{hash}(H_{sk}; sk)$.
2: Partition the tokens in y into groups $Y_0, \ldots, Y_{\hat{n}-1}$ according to their corresponding positions.
3: Initialize the count array $\text{COUNT}_p^{(1)}$ for each position to an all-zero array of length $2^{\hat{m}}$, where $p = \{0, \ldots, \hat{n}-1\}$.
4: Set $H_p^{(1)} = \text{Hash}(0, s_H)$ for $p = \{0, \ldots, \hat{n}-1\}$.
5: **for** $p = 0, \ldots, \hat{n}-1$ **do**
6: /* Each iteration is independent and can be executed in parallel */
7: **for** $i = 1, 2, \ldots, N_f$ **do**
8: Build $\mathcal{C}_{hash}(\text{COUNT}_p^{(i)}, s_H, H_p^{(i)})$ for consistency check.
9: **for** $j = 1, 2, \ldots, N_t$ **do**
10: Invoke Lines 2 to 6 in Alg. 2 to build the constraints $\mathcal{C}_{hash}$ and $\mathcal{C}_{flg}$.
11: **end for**
12: Accumulate the increment of $\text{COUNT}_p^{(i)}$ to $\text{COUNT}_p^{(i+1)}$ by

$$\text{COUNT}_p^{(i+1)}[j] = \text{COUNT}_p^{(i)}[j] + flg_j^p$$

and establish the addition constraints.
13: /* The subsequent content is similar to Lines 8 to 12 in Alg. 1. We omit the details here. */
14: **end for**
15: /* The subsequent content is similar to Lines 14 to 16 in Alg. 1. We omit the details here. */
16: **end for**
17: Return $\Pi = \{\pi_1^p, \pi_2^p\}$ for $p = \{0, \ldots, \hat{n}-1\}$.

A Recursive ZKP for SynthID-Text and Segment-Watermark

We give the details of detection proof generation using recursive ZKP for SynthID-Text and Segment-Watermark in Alg. 4 and Alg. 5, respectively.

B Multi-Bit Payload Scalability

We evaluate the payload-size scalability of the multi-bit Segment-Watermark instantiation by varying the number of bits $\hat{m}$ carried by each message position, with the results shown in Table 13.

Table 13: Scalability of the multi-bit Segment-Watermark instantiation with larger payload bits. Each entry reports setup/prove time in seconds under $\hat{n} = 6$.

Payload bits	240 tokens	960 tokens	1920 tokens
$\hat{m} = 5$	22.58 / 18.02	23.52 / 58.21	24.21 / 106.05
$\hat{m} = 7$	78.41 / 60.58	161.51 / 187.52	167.57 / 319.33

C Security Analysis of PVMark

We claim that Alg. 1, Alg. 2, Alg. 3, Alg. 5 and Alg. 4 are zero knowledge arguments by definition), which generate correctness proofs for the watermark detection process without revealing the secret key. We provide a security proof of Alg. 1 in Thm. C.1. The proofs for the other four algorithms are similar, and we omit them here due to limited space.

Theorem C.1. *Alg. 1 is a zero knowledge argument.*

Proof. **Completeness.** As explained in Sec. 6, the constraints $\mathcal{C}$ are satisfied if the public input u and the private input w are the correct parameters. Therefore, the correctness of Alg. 1 follows the correctness of zero knowledge arguments (Groth16, PlonK, halo2, or Nova, depending on the implementation), which are proven in previous works.

Soundness. For any probabilistic polynomial time (PPT) dishonest prover $\mathcal{P}^*$, there exists a PPT extractor ε such that given access to the executing process of Alg. 1 and public input u, can extract a witness $w^* = \{sk^*\}$ that

$$\Pr[(u; w^*) \notin \mathcal{C} \wedge \mathcal{V}(u, \pi^*, pp) = 1] \leq \text{negl}(\lambda),$$

according to the soundness property of the zero knowledge arguments. Furthermore, the soundness of the protocol effectively mitigates the **fake random seed attack**. In this scenario, a dishonest owner may attempt to fabricate random seeds to manipulate the ZKP generation. However, such malfeasance is precluded by the soundness property of the system. Given the one-way and collision-resistant nature of the employed cryptographic hash function, it is computationally infeasible for an adversary to identify a counterfeit seed that satisfies the verification constraints. Consequently, the probability of a successful attack is rendered negligible.

Zero knowledge. For every verifier $\mathcal{V}$, there exists a simulator $\mathcal{S}$ such that given oracle access to the public input $u = \{y, \psi, \gamma|\mathbb{F}|/\frac{|\mathbb{F}|}{2}, |y_G|/S_g, \xi\}$, $\mathcal{S}$ is able to simulate the view of $\mathcal{V}$ in verifying proof Π generated by Alg. 1. The details are as follows.

With oracle access to $u = \{y, \psi, \gamma|\mathbb{F}|/\frac{|\mathbb{F}|}{2}, |y_G|/S_g, \xi\}$, and the simulator $\mathcal{S}_1$ of the zero-knowledge argument introduced in [12, 17, 25], we construct the simulator $\mathcal{S}$ as following: 1) $\mathcal{S}$ initializes the relevant setup parameters and sends the public input u to $\mathcal{V}$; 2) $\mathcal{S}$ establishes the constraints for the computation of hash functions, the setting of flg and summation operations; and 3) $\mathcal{S}$ calls $\mathcal{S}_1$ to simulate the partial view of the zero knowledge argument for the constraints and generates π. Then $\mathcal{S}$ sends the proof $\Pi = \{u, \pi\}$ to $\mathcal{V}$.

To prove zero-knowledge, step 1 and step 2 are obviously indistinguishable in both real world and ideal world as the input $u = \{y, \psi, \gamma|\mathbb{F}|/\frac{|\mathbb{F}|}{2}, |y_G|/S_g, \xi\}$ and the form of constraints are public. Step 3 of worlds is indistinguishable because of the zero knowledge property of the zero knowledge arguments proven in [12, 17, 25], which completes the proof. □