

# PVMark: Enabling Public Verifiability for LLM Watermarking Schemes

Haohua Duan, Liyao Xiang\*, Member, IEEE, and Xin Zhang

**Abstract**—Watermarking schemes for large language models (LLMs) have been proposed to identify the source of the generated text, mitigating the potential threats emerged from model theft. However, current watermarking solutions hardly resolve the trust issue: the non-public watermark detection cannot prove itself faithfully conducting the detection. We observe that it is attributed to the secret key mostly used in the watermark detection — it cannot be public, or the adversary may launch removal attacks provided the key; nor can it be private, or the watermarking detection is opaque to the public. 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 hinges upon the proof of ‘correct execution’ of watermark detection on which a set of ZKP constraints are built, including mapping, random number generation, comparison, and summation. We implement multiple variants of PVMark in Python, Rust and Circom, covering combinations of three watermarking schemes, three hash functions, and four ZKP protocols, to show 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, promising to be deployed in practice.

**Index Terms**—LLM watermarking, zero-knowledge proof, public verifiability.

## I. INTRODUCTION

In recent years, large language models (LLMs) have made significant impact to the community by generating high-quality texts akin to human-written ones. However, this advancement also brings many real-world threats such as the spread of generated fake news [1], misuse of LLMs in academic misconduct [2], etc. To achieve responsible AI, it is important to trace the provenance of the generated texts by LLMs.

Towards this end, various watermarking schemes [3]–[12] are proposed for LLMs. The mainstream of the watermarking schemes embeds the watermark by altering the probability distribution of the generated tokens depending on the context and owner’s secret key. The watermark is later extracted by detecting the statistical significance in the word distribution of the suspected text. However, such watermarking schemes face a trust crisis in practical applications. As shown in Figure 1, when the LLM owner proclaims its right against unauthorized use, it may face questions from a third party, e.g., the court, about the legitimacy of the watermark, due to the opaque watermark detection involving a secret key. In such a case,

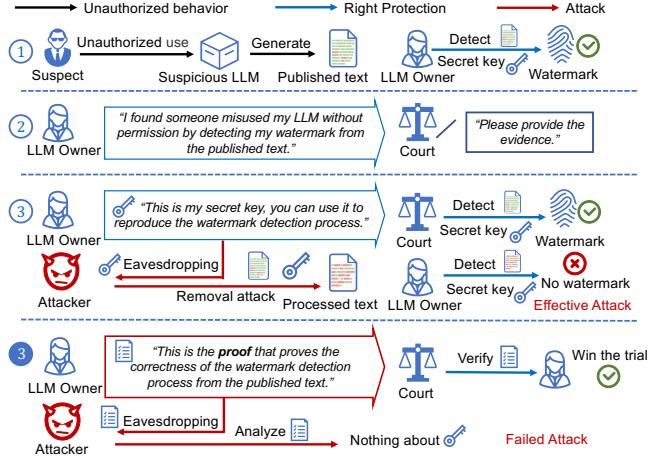


Fig. 1: From top to bottom: ① The LLM owner detects unauthorized use of the LLM by watermark detection. ② The LLM owner sues the suspected model and the court requests evidence. ③ The LLM owner’s secret key gets exposed in answering the court’s request. ④ The LLM owner provides the proof as evidence under PVMark thus preventing the attack.

the owner has to risk its secret key to third-party detection, while the third party may be non-confidential allowing an eavesdropper to acquire the secret key, threatening the validity of the watermark. The root cause of this problem is the use of a secret key which cannot be public — otherwise putting the watermarked model under the threat of removal or ambiguity attacks; nor private — for hardly convincing others that the detection is legitimate. Overall, LLM watermarking schemes lack *public verifiability*.

Previous efforts have been devoted to resolve the issue. Fairoze *et al.* attempt to use a private key for watermark embedding and a public key for detection [13]. However, the detection relies much on extracted features of the generated texts, which can be exploited by adversaries. Liu *et al.* address the problem by training two separate neural networks as watermark embedding and detection modules respectively [14], but the exposure of the detection network still enables the adversary to train a reverse network to remove the watermark. Kirchenbauer *et al.* propose a private detection scheme, allowing only authorized parties to access the detection APIs [3], yet without convincing others of the faithful execution of APIs.

To resolve the dilemma, we design a plugin named *PVMark*, which augments the state-of-the-art LLM watermarking schemes with *public verifiability* while preserving the original schemes’ properties of effectiveness, fidelity, and robustness.

Specifically, without revealing the secret key, the owner or an authorized party can perform a publicly verifiable detection to convince others that the detection is faithfully done, thus preventing any attack raised by theft of secret key.

*PVMark* achieves public verifiability by integrating zero knowledge proof (ZKP) into watermark detection. ZKP is a protocol in which the prover can convince the verifier that a given statement is true, without conveying to the verifier any information beyond the fact that the statement is true. It is aligned with our goal of verifying the watermark detection is correctly done yet without revealing the secret key.

To instantiate the idea, we select three representative LLM watermarking schemes, i.e., KGW [3], SynthID-Text [4], and Segment-Watermark [12], to implement *PVMark*. We found these schemes share a common token-sampling-alteration framework: at each sampling round, all candidate tokens are randomly divided into different groups based on the secret key and the watermark. Then tokens in the ‘green’ group are given a slightly higher probability to be sampled as the output. Hence the statistical attributes of the output can be used to extract the watermark.

Our core idea is to provide evidence that the statistical results of green tokens in the produced text are faithfully collected without exposing the secret key, thereby demonstrating that the text indeed contains watermarks. Hence two criteria need to satisfy: the watermark is extracted from the text, and the proof of the detection passes verification. In applying ZKP to watermarking, we mainly resolve two technical challenges: 1) how to model the watermark detection by circuits on which ZKP protocols run, and 2) how to minimize the overhead introduced by ZKP thereby improving detection efficiency.

To tackle challenge 1, we decompose the watermark detection into four components for reconstruction — mapping table check, hashing, comparison, and summation, replacing the original components by ZKP-friendly ones. For example, the pseudorandom function (PRF) instantiated by a modular operation-based linear congruential generator is inefficient for ZKP, which is substituted by cryptographic hash functions. The identification of ‘green’ tokens is reconfigured as comparing the generated random numbers against a fixed threshold. We show that the reconstruction of the watermarking schemes not only preserves the watermark properties, but also tailor them to ZKP protocols.

Apart from reconstruction, we propose multiple approaches to improve the efficiency of *PVMark* (addressing challenge 2). Since hash functions are heavily used in detection, we combine them into three-to-one hash functions for more succinct constraints. Observing the watermark detection is a repeated procedure on a sequence of tokens, we treat the detection on a subsequence of tokens as an instance and reduce checking of multiple instances into checking one by folding [15]. Hence we reduce the ZKP overhead for proving multiple instances into proving one final instance and the folding steps, significantly mitigating the 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.

## II. RELATED WORK

Currently, multiple schemes are proposed for LLM watermarking. One is retrieval-based schemes [16] that require accessing and storing all the texts generated during the interaction process of LLM. This largely destroys the privacy of users using LLM. The second is post hoc detection schemes [17]–[19] that usually train a machine learning model to distinguish human-written text from LLM-generated text. These schemes require a lot of computing resources and the performance is not good enough [20]. The third is represented by KGW scheme [3]. The high-level idea of KGW is to divide the vocabulary into green and red token lists, and modify logits to shift the distribution of candidate tokens toward the green lists. This approach allows watermark embedding without requiring any LLM fine-tuning, making it general and less costly to deploy.

Many watermarking schemes based on probability shifting have been proposed subsequently. Lee *et al.* [5] and Wang *et al.* [6] reduce the impact of logits modification on text quality by only watermarking high-entropy tokens. Yoo *et al.* propose to assign different message bits to distinct positions within the text to increase the payload of the watermark in [7]. Qu *et al.* propose using dynamic programming algorithms to allocate tokens to each position of the message to be embedded in a balanced manner to improve the accuracy of watermark detection. For robustness, Kirchenbauer *et al.* study the effects of different hash strategies on watermarking robustness in [8]. Zhao *et al.* demonstrate that using a globally fixed vocabulary partitioning scheme provides better resistance against removal attacks in [9]. Hu *et al.* [10] and Wu *et al.* [11] propose an unbiased weighting method to improve the imperceptibility of KGW. Dathathri *et al.* propose Tournament sampling algorithm to improve watermark detection rate while preserving text quality in [4].

However, the aforementioned schemes are only suitable for private watermark detection scenarios and cannot be applied to public detection, as exposing the detector would face various threats.

To address the public verifiability of watermark, Fairoze *et al.* utilize the asymmetric encryption to embed the watermark using a private key and detect it by a public key [13]. However, this approach relies on text features for watermark detection, which can also be exploited by adversaries for spoofing attacks. Liu *et al.* propose training two additional neural network models in [14], one for watermark embedding and another for watermark detection. But, when the window size, which decides the partition of vocabulary for the current token, is relatively small, the method would suffer from reverse engineering

attack, i.e., reverse training of the embedding network from the detection network, threatening the robustness of the system.

### III. PRELIMINARY

#### A. Zero Knowledge Proof

Zero knowledge proofs (ZKPs) is a tool whereby a computationally unbounded party (the prover) can prove to another party (the verifier) that a given statement is true, while avoiding conveying any additional information apart from the fact that the statement is indeed true. ZKP ensures that if the prover has the correct private input  $w$  and public input  $u$ , it can convince the verifier of the statement's validity with a very high probability (completeness property), without revealing anything about  $w$  (zero-knowledge property). Conversely, if the prover does not have  $w$ , the verifier will reject the proof with a very high probability (soundness property). The formal definitions of zero knowledge arguments can be found in supplementary file.

In the context of ZKPs, an arithmetic circuit is a system of equations that models a problem and on which ZKP protocols run to verify. Common circuits include GKR-based ones, rank-1 constraint system (R1CS), PLONKish arithmetization, etc. For simplicity, we use PLONKish as an example type of circuit to introduce our system. From a high-level perspective, PLONKish arithmetization encompasses the numerical values of both public and private inputs, along with equations that describe the relationships among these values, denoted as constraints. The form of constraints used by the standard PLONKish is as follows:

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

where  $s_L, s_R, s_O, s_M, s_C \in \mathbb{F}$  are selectors, and  $x_L, x_R, x_O, c \in \mathbb{F}$  denotes the left input, the right input, the output and the constant value, respectively.

**Recursive ZKP.** Nova [15] is an effective recursive zero-knowledge protocol from folding schemes to realize incrementally verifiable computation (IVC). IVC refers to recursively verifiable computations of the form

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

where  $F, w, u$  and  $\eta$  denote the computation function, private input, public input and the number of computations, respectively. Nova can fold two relaxed R1CS instances into one and the relaxed R1CS can be expressed as follows:

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

where  $A, B, C$  are matrices containing ZKP circuits information,  $\mu$  is a scalar and  $\nu$  is used to store redundant items. The  $\odot$  denotes the hadamard product.  $Z = [u, w, 1]$  is the instance vector including the witness  $w$ , public input  $u$  and the constant 1. Two relaxed R1CS instance  $(Z_1, \mu_1, \nu_1)$  and  $(Z_2, \mu_2, \nu_2)$  can be folded into one instance  $(Z, \mu, \nu)$  as follows:

$$\begin{aligned} \nu &= \nu_1 + r \cdot (AZ_1 \odot BZ_2 + AZ_2 \odot BZ_1 \\ &\quad - \mu_1 \cdot CZ_2 - \mu_2 \cdot CZ_1) + r^2 \cdot \nu_2, \end{aligned} \quad (4)$$

$$\mu = \mu_1 + r \cdot \mu_2, \quad (5)$$

$$Z = Z_1 + r \cdot Z_2, \quad (6)$$

where  $r$  denotes randomness. It suffices to provide proofs for the correctness of Eq. (4), (5), (6), and the folded instance,

eliminating the need for proofs of the original two instances. With careful design, savings are achieved if the constraints for a single fold are fewer than those for an individual instance.

#### B. Language Model Watermarking

A language model (LM) is a type of machine learning model trained to predict the next most appropriate word to fill in a blank space in a sentence or phrase, based on the given text. Given an input prompt  $x = [x_0, \dots, x_{m-1}]$  of length  $m$ , a language model  $\mathcal{M}$  is a function that iteratively computes the logit scores  $l_j^{(i)}$  ( $0 \leq i \leq n-1, j \in |V|$ ) of the next  $n$  tokens over the vocabulary  $V$ . These logits are transformed into a probability distribution  $p^{(i)}$  via the softmax and the LM  $\mathcal{M}$  samples the next  $n$  tokens  $y = [y_0, \dots, y_{n-1}]$  from  $p^{(i)}$ . We overload the syntax  $y_i$  to denote the index of the corresponding token in the vocabulary without causing confusion.

Language model watermarking is a technique that allows the model owner to embed identifiable messages known as watermarks within the sequence  $y$  generated by a watermarked LM  $\hat{\mathcal{M}}$ . We give the formal definition of language model watermarking in the supplementary file due to space limitation.

**Common LLM watermarking frameworks.** Existing LLM watermarking schemes based on probability shifting can be categorized into zero-bit watermarking schemes [3]–[11], [21] and multi-bit watermarking schemes [12] based on their capability to embed messages into LLM-generated text. The former type only injects features to differentiate AI-generated text from human-produced text without embedding any explicit information, whereas the latter can embed multi-bit metadata — such as a model owner's identity — into text for more precise identification and traceability.

Despite differences in technical details, these two categories of probability-shifting-based watermarking schemes share a common high-level framework. We select representative works in zero-bit watermarking schemes, namely KGW [3] and SynthID-Text [4], as well as Segment-Watermark [12] in multi-bit watermarking schemes to showcase these watermarking schemes.

The watermark embedding process is summarized in Table I 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  $i$ , the first step is to generate a random seed  $sd_i$  based on the secret key  $sk$  and contextual information  $y_\psi$  where  $\psi$  denotes the recent context window width. This step in multi-bit watermarking additionally requires the mapping of tokens into positions ( $p$ ) and the position index is used to pick the corresponding message bit ( $msg[p]$ ) to generate the random seed. The seed  $sd_i$  is then employed to partition candidate tokens into groups for the second step. In Step 3, the logits of tokens in the selected groups are adjusted accordingly by means of adding a bias to increase the probability of being sampled. In essence, this probability-shifting mechanism inclines the LLM to preferentially output tokens from the selected group, thereby embedding watermark information into the LLM-generated text. In particular for SynthID-Text, the probability shifting is

TABLE I: The watermark embedding process of zero-bit and multi-bit LLM watermarking schemes. Steps requiring adjustment for PVMark are marked in brackets.  $\text{PRF}(\cdot)$  denotes a pseudo-random function.

	<b>Step 1:</b> Random seed generation.	<b>Step 2:</b> Vocabulary partitioning.	<b>Step 3:</b> Probability shifting and sampling.
<b>KGW [3]</b>	Compute random seed $sd_i \leftarrow \text{PRF}(sk, \mathbf{y}_\psi)$ . $\Rightarrow$ [Adjustment for PVMark: Compute random seed $sd_i \leftarrow \text{Hash}(sk, \mathbf{y}_\psi)$ .]	<p>a) Assign random number <math>g^{(y_j)} = \text{PRF}(sd_i, y_j)</math> to each token <math>y_j \in V</math>; <math>\Rightarrow</math> [Adjustment for PVMark: Assign random number <math>g^{(y_j)} = \text{Hash}(sd_i, y_j)</math> to each token <math>y_j \in V</math>]</p> <p>b) Sort <math>V</math> by all the random numbers <math>g^{(y_1)}, \dots, g^{(y_{ V })}</math> corresponding to token <math>y_1, \dots, y_{ V } \in V</math> and designate the top <math>\gamma V , \gamma \in (0, 1)</math> tokens in the sorted <math>V</math> as green list <math>G</math>. <math>\Rightarrow</math> [Adjustment for PVMark: Partition the all candidate tokens as follows:</p> $\text{Candidate token } y_j \in \begin{cases} G, & g^{(y_j)} < \gamma \cdot  \mathbb{F}  \\ R, & g^{(y_j)} \geq \gamma \cdot  \mathbb{F}  \end{cases}$ <p>for <math>j \in  V </math>, where <math>R</math> and <math>\mathbb{F}</math> denotes the red list and finite field, respectively.]</p>	Add a bias $\delta$ to logits of all tokens in $G$ and compute the new probability distribution $\hat{\mathbf{p}}^{(i)}$ using new logits over $V$ . Sample $y_i$ from $\hat{\mathbf{p}}^{(i)}$ and append $y_i$ to $\mathbf{y}$ .
<b>SynthID-Text [4]</b>		<p>Generate top K candidate set <math>V</math> for this round and assign <math>\xi</math> random numbers <math>g_k^{(y_j)} = \text{PRF}(sd_i, y_j, k)</math> to each token <math>y_j \in V</math> for <math>k \in \{1, \dots, \xi\}</math>; <math>\Rightarrow</math> [Adjustment for PVMark: Generate top <math>K = 2^\xi</math> candidate set <math>V</math> for this round and assign <math>\xi</math> random numbers to each token <math>y_j \in V</math> by</p> $g_k^{(y_j)} = \begin{cases} 1, & \text{Hash}(sd_i, y_j, k) <  \mathbb{F} /2, \\ 0, & \text{Hash}(sd_i, y_j, k) \geq  \mathbb{F} /2, \end{cases}$ <p>for <math>k \in \{1, \dots, \xi\}</math>; ]</p> <p>For each <math>y_j \in V</math>, set <math>y_j^{(0)} \leftarrow y_j</math>.</p>	<pre> <b>for</b> <math>k = 1, \dots, \xi</math> <b>do</b>   <b>for</b> <math>j = 0, \dots, 2^{\xi-k} - 1</math> <b>do</b>     <math>J = [y_{2j}^{(k-1)}, y_{2j+1}^{(k-1)}]</math> (may contain repeats).     <math>J^* = [y \in J \text{ s.t. } g_k^{(y)} = \max_{y' \in J} g_k^{(y')}</math> (may contain repeats).     Sample <math>y_j^{(k)}</math> from the uniform distribution of <math>J^*</math>.   <b>end for</b> <b>end for</b> Append <math>y_0^\xi</math> to <math>\mathbf{y}</math>. </pre>
<b>Segment-Watermark [12]</b>	<p>a) Find position index <math>p_i = M[\mathbf{y}_\psi]</math>;</p> <p>b) Compute random seed <math>sd_i \leftarrow \text{PRF}(sk, \mathbf{y}_\psi, msg[p_i])</math>. <math>\Rightarrow</math> [Adjustment for PVMark: Compute random seed <math>sd_i \leftarrow \text{Hash}(sk, \mathbf{y}_\psi, msg[p_i])</math>.]</p>	The same as <b>KGW</b> .	The same as <b>KGW</b> .

TABLE II: The watermark detection process of zero-bit and multi-bit LLM watermarking schemes. Steps requiring adjustment for PVMark are marked in brackets.

	<b>Step 1:</b> Compute random seed	<b>Step 2:</b> Check and count green tokens	<b>Step 3:</b> Compute score / message
<b>KGW [3]</b>	Compute random seed $sd_i \leftarrow \text{PRF}(sk, \mathbf{y}_\psi)$ . $\Rightarrow$ [Adjustment for PVMark: Compute random seed $sd_i \leftarrow \text{Hash}(sk, \mathbf{y}_\psi)$ .]	<p>a) Similar to Step 2 of the watermark embedding process, assign a random number to each token in the vocabulary using <math>sd_i</math> and reorder the vocabulary to find <math>G_i</math>; <math>\Rightarrow</math> [Adjustment for PVMark: Compute the random number <math>g^{y_i} = \text{Hash}(sd_i, y_i)</math> corresponding to the current token <math>y_i</math>]</p> <p>b) Count the number of green tokens <math> y _G += \mathbb{1}(y_i \in G_i)</math> where <math>\mathbb{1}(\cdot)</math> is the indicator function. <math>\Rightarrow</math> [Adjustment for PVMark: Count the number of green tokens <math> y _G += \mathbb{1}(g^{y_i} &lt;  \mathbb{F} )</math>.]</p>	$\text{Score}(\mathbf{y}) = ( \mathbf{y} _G - \gamma n) / \sqrt{n\gamma(1-\gamma)}$
<b>SynthID-Text [4]</b>		<p>a) Compute <math>\xi</math> random number <math>g_k^{y_i}</math> corresponding to the current token <math>y_i</math> where <math>k \in \{1, \dots, \xi\}</math>; <math>\Rightarrow</math> [Adjustment for PVMark: Compute <math>\xi</math> random number corresponding to the current token <math>y_i</math> by</p> $g_k^{(y_i)} = \begin{cases} 1, & \text{Hash}(sd_i, y_i, k) <  \mathbb{F} /2, \\ 0, & \text{Hash}(sd_i, y_i, k) \geq  \mathbb{F} /2, \end{cases}$ <p>for <math>k \in \{1, \dots, \xi\}</math>; ]</p> <p>b) Update <math>S_g</math>: <math>S_g += \sum_{k=1}^{\xi} g_k</math>.</p>	$\text{Score}(\mathbf{y}) = \frac{S_g}{(n-\psi)\cdot\xi}$
<b>Segment-Watermark [12]</b>	<p>a) Find position index <math>p_i = M[\mathbf{y}_\psi]</math>;</p> <p>b) Compute all possible random seeds <math>sd_i^j \leftarrow \text{PRF}(sk, \mathbf{y}_\psi, j)</math> for <math>j = 0, 1, \dots, 2^m - 1</math>. <math>\Rightarrow</math> [Adjustment for PVMark: Compute all possible random seeds <math>sd_i^j \leftarrow \text{Hash}(sk, \mathbf{y}_\psi, j)</math> for <math>j = 0, 1, \dots, 2^m - 1</math>.]</p>	<p><b>for</b> <math>j = 0, 1, \dots, 2^m - 1</math> <b>do</b></p> <p>Assign a random number to each token in the vocabulary using <math>sd_i^j</math> and reorder the vocabulary to find <math>G_i^j</math>; <math>\Rightarrow</math> [Adjustment for PVMark: Compute the random number <math>g^{y_i} = \text{Hash}(sd_i^j, y_i)</math> corresponding to the current token <math>y_i</math>];</p> <p>Update COUNT: <math>\text{COUNT}[p_i][j] += \mathbb{1}(y_i \in G_i)</math>. <math>\Rightarrow</math> [Adjustment for PVMark: Update COUNT: <math>\text{COUNT}[p_i][j] += \mathbb{1}(g^{y_i} &lt; \gamma \cdot  \mathbb{F} )</math>.]</p> <p><b>end for</b></p>	$msg[p] \leftarrow \arg \max_{j \in [0, 2^m - 1]} \text{COUNT}[p][j]$ <p>for <math>p = 0, 1, \dots, \hat{n} - 1</math>.</p>

applied implicitly through Tournament sampling where every two candidates compare their  $g$  values to enter the next-tier competition until the final winner is selected. It has been proved that the Tournament sampling in fact raises the probability of the winner and lowers the probability of the other through each round of competition.

The watermark detection processes for zero-bit and multi-bit schemes exhibit subtle distinctions, shown in Table II. For zero-bit schemes, the random seed derived from the secret key  $sk$  and context  $\mathbf{y}_\psi$  is used to determine the token grouping for each round. The number of tokens in the given text  $\mathbf{y}$  that belong to the selected group is counted as  $|\mathbf{y}|_G$ . A watermark score  $\text{Score}(\mathbf{y})$  is computed based on  $|\mathbf{y}|_G$ , and this score is

compared against a threshold  $\tau$  to identify the being of the watermark. In contrast, multi-bit schemes involve iterating over all possible watermark message values in calculating random seeds, alongside the secret key and context. By analyzing token group statistics associated with different watermark message values, the detection algorithm selects the highest one as the decoded message.

#### IV. PROBLEM STATEMENT AND PVMARK OVERVIEW

In this section, we point out the threats that the common LLM watermarking frameworks face in ownership verification or generated content provenance tracing. We formally define the problem and then give a solution overview.

**Threat model.** In ownership verification and provenance tracing, the watermarking frameworks involve a watermark embedding party, usually the model owner, and a watermark detection party. The detection party extracts the watermark from the watermarked text as evidence of the text origins. In previous works [3]–[12], [21], the detection parties are mostly the embedding parties, as they own the detection key for watermark detection. However, the confidential detection process would *reduce its credibility* to the public. If the detection party is acted by a third party, the party has to keep the detection key confidential. The leakage of the detection key to a malicious party would tempt an adaptive attack to remove the watermark from the watermarked content, known as *removal attacks*. This implies that the aforementioned LLM watermarking frameworks lack *public verifiability*.

**Motivation.** We observe that the lack of public verifiability is due to the key  $sk$  used in the embedding and the detection phases. The  $sk$  is crucial to be kept secret as its exposure would leak the vocabulary partition for watermark embedding, allowing adversaries to identify ‘green tokens,’ and thus intentionally replacing them to remove the watermark.

Our intuition is that,  $sk$  cannot be made public while the watermark detection should be transparent. To resolve the dilemma, we propose a plugin *PVMark* that enables public verification of LLM watermarking without leaking  $sk$ : it provides a correctness proof of the watermark detection to any verifier, which circumvents the direct exposure of the detection process. Specifically, by integrating ZKP into watermarking detection, *PVMark* can hide  $sk$  (zero-knowledge property) while providing evidence that the detection is correctly performed by the relevant party (completeness property). Meanwhile, the probability of cheating to pass verification is negligible (soundness property). Even though the proof only serves the detection, the embedding procedure requires adaptation to the proof.

**Problem statement.** Formally, four parties are involved in *PVMark*: (1) *watermark embedding party*  $\mathcal{E}$  who embeds the watermark into  $\mathcal{M}$  to obtain  $\hat{\mathcal{M}}$ ; (2) *watermark detection party*  $\mathcal{D}$  who has access to the watermark detector,  $sk$  corresponding to  $\hat{\mathcal{M}}$ , and a text sequence  $\mathbf{y}$ .  $\mathcal{D}$  extracts watermark from  $\mathbf{y}$  while acting as the *prover* to prove the claim that ‘ $\mathbf{y}$  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 exploits  $\hat{\mathcal{M}}$  to generate unauthorized text with watermark removed.

**Use case.** *Ownership verification.* The model owner  $\mathcal{E}$  trains a language model  $\mathcal{M}$  and embeds its watermark to get  $\hat{\mathcal{M}}$ . If any unauthorized party is suspected to illegally possess  $\hat{\mathcal{M}}$  and generate text  $\mathbf{y}$  by it, the watermark detection party  $\mathcal{D}$ , typically the owner or an authorized party, could provide the evidence by detecting the watermark from  $\mathbf{y}$ , generating the corresponding correctness proof and submitting it to the verifier  $\mathcal{V}$ . It is confirmed that the unauthorized party illegally possesses  $\hat{\mathcal{M}}$ , should the watermark be detected and the proof be verified.

*Provenance tracing.* The owner  $\mathcal{E}$  generates content  $\mathbf{y}$  with watermark embedded. Someone raises concern on  $\mathbf{y}$  and tracks the origin of the text. The authorized party  $\mathcal{D}$  detects watermark belonging to  $\mathcal{E}$  and provides evidence to  $\mathcal{V}$  that the detection is faithfully done.  $\mathcal{E}$  cannot deny the results supported by evidence.

In both use cases, the embedding and detection algorithms are publicly known, except for the secret key  $sk$ .  $\mathcal{D}$  uses  $sk$  in detection and proof generation without revealing it.

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

**Definition IV.1.** *PVMark* is a plugin that enables public verifiability on a language model watermarking scheme. It consists of a triple of setup, embedding and detection algorithms:

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

In previous work [3]–[12], [21], the properties of watermarking schemes have predominantly focused on *effectiveness* and *fidelity*, meaning the watermark should be embedded and detected with high accuracy while maintaining decent text quality, and *robustness*, which hinders adversaries from removing the watermark from watermarked text or forging the watermark into unwatermarked text. We claim that *PVMark* can augment watermarking schemes with an additional property of *public verifiability* without undermining the aforementioned attributes — specifically, the property allows any third party to check the integrity of the watermark detection result.

It is noteworthy that Pang *et al.* point out an inherent conflict between robustness and public detectability for watermarking systems in [22]. 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.

From the following section on, we will introduce *PVMark* in detail.

## V. 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 (see experimental results in the supplementary file).

### A. Hash-Based Vocabulary Partitioning

A key step in watermark embedding is to assign random ranks to candidate tokens in realizing the random partition of

vocabulary into green and red lists (Step 2 in Table I). Green tokens are selected with a probability higher than random chance.

The original watermarking schemes primarily employ two methods for vocabulary partitioning. The first is random permutation, utilized in KGW [3] and Segment-Watermark [12]: it randomly permutes all the candidate tokens in the vocabulary and marks the first  $\gamma \cdot |V|$  indexed tokens as the green list and the rest as the red list. Random permutation requires a mapping operation from random numbers to the range of the vocabulary size, which involves ZK-unfriendly modulo operations.

The second is the tournament method, adopted by SynthID-Text [4]: 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  $\xi$  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  $g_1, \dots, g_\xi$ . 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  $r = \{r_1, \dots, r_{|V|}\}$  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 I. 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 I). 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. VIII-B).

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 II.

### B. Selection of the Hash Function

We select cryptographic hash functions as PRFs since they are high-quality pseudorandom number generators following the guidelines NIST SP 800-90A [23], instead of non-cryptographic pseudorandom number generation algorithms such as Linear feedback shift register [24], Blum Blum Shub [25], Mersenne

Twister [26], Xorshift [27], etc. Although the non-cryptographic ones are faster by bit operations, they are not ZKP-friendly for incurring high costs in the verification of bitwise operations.

The most commonly used cryptographic hash functions include SHA256, Keccak256, BLAKE2, which take inputs of arbitrary length and map them to a fixed 256-bit output. These hash functions contain many bit operations which involve high ZKP-related costs. For instance, in verifying SHA256, the significant cost arises from its compression function that contains bit decompositions. Other hash functions are more ZK-friendly, including MiMC hash [28], Poseidon hash [29] and Poseidon2 hash [30]. 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. VIII-B. 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.

## VI. BUILDING BLOCKS OF PVMARK

To prove watermark detection, it is required for *PVMark* to generate proofs for the three steps in Table II. We observe that in Step 3 for computing the score/message, the only unknowns are  $y_G$ ,  $S_g$ , and COUNT, while other quantities are public. Hence, 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. Therefore, *PVMark* only needs to prove the correctness of  $y_G$ ,  $S_g$ , and COUNT, i.e., generate correctness proofs for Step 1 and Step 2, yet without exposing  $sk$  or the green list  $G$ . Next, we will present in detail the arithmetization of the two steps by the example of PLONKish.

### A. Arithmetize Step 1 in Table II

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

**Hash functions.** The constraints  $C_{hash}(\text{Output}; \text{Inputs})$  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  $H_{\text{mimc}}(X) = (f_{\phi-1} \circ f_{\phi-2} \circ \dots \circ f_0)(X) + k$  where  $\phi$  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  $s_{\text{Pow}}$  to rewrite Eq. (1) as

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

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)  $\phi$  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  $s_{\text{Pow}}$  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  $y_\psi$  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  $\text{msg}[p]$  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  $\mathcal{C}_{\text{mem}}$  are as follows:

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

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

### B. Arithmetize Step 2 in Table II

In Step 2, we need to compute the corresponding random number for the current token  $y$ , compare this random number with a fixed threshold, and perform an accumulation operation if the token is a green token. 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  $\mathcal{C}_{\text{bit}}$  [31] can be used to check whether  $X_3$  is in this range. The constraints for comparison  $\mathcal{C}_{\text{comp}}(X_1 < X_2)$  can be written as follows:

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

where  $\mathbf{b}_{X_3} = [b_0, b_1, \dots, b_{N-1}]$  denotes the binary representation of  $X_3$ .  $\mathcal{C}_{\text{comp}}(X_1 < X_2) = 0$  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  $\mathcal{C}_{\text{bit}}(\cdot)$  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  $b_i$ s 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 predefined 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  $\text{flg}_i \in \{0, 1\}$  to indicate whether token  $y_i$  belongs to the green list  $G_i$ . The following constraint  $\mathcal{C}_{\text{flg}}$  verifies the assignment of  $\text{flg}_i$ :

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

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  $\mathcal{C}_{\text{hash}}$ . Then we introduce a summation selector  $s_{\text{Sum}}$  and rewrite Eq. (1) as

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

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

## VII. 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.

---

**Algorithm 1** Proof Construction for KGW / SynthID-Text
 

---

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

**Output:** Proof  $\Pi$ .

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

---

#### A. Verifying Zero-& Multi-Bit Watermarks

*PVMark* is applied right after the watermark detection phase by prover  $\mathcal{D}$ , requiring the detection results  $|\mathbf{y}|_G$ ,  $S_g$ , COUNT and other constants as inputs. The output of proof construction is a proof  $\pi$  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  $\pi$  for all the constraints and releases  $\pi$  and the public inputs to  $\mathcal{V}$  to carry out proving and verification following ZKP protocols.

**Combining hash functions.** In the original KGW scheme,  $\psi$  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 II). 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.

---

**Algorithm 2** Proof Construction for Segment-Watermark
 

---

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

**Output:** Proof  $\Pi$ .

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

---

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,  $\psi$  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,  $\mathcal{C}_{hash}(sd_i; sk, \mathbf{y}_\psi)$  is defined as the set

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

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

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

#### B. 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  $\mathbf{y} = [y_0, \dots, y_{n-1}]$  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

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

Within  $F$ , we compute the random number corresponding to  $y_i$  and compare it with the pre-selected threshold  $\gamma|\mathbb{F}|$ . If the

---

**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  $\Pi$ .

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

---

random number is less than the threshold,  $y_i$  is a green token, and we set  $|\mathbf{y}|_G^{(i+1)} = |\mathbf{y}|_G^{(i)} + 1$ . After  $n - 1$  iterations,  $|\mathbf{y}|_G^{(n)}$  represents the count of green tokens in text  $\mathbf{y}$ .

Given the incremental calculation, we can efficiently generate zero-knowledge proofs using Nova's folding scheme, turning the detection of each token in  $\mathbf{y}$  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  $|\mathbf{y}|_G^{(i)}$  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  $|\mathbf{y}|_G^{(i)}$  instead of the value. This modifies Eq. (12) to

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

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  $|\mathbf{y}|_G^{(i)}$  of this round agrees with the output from the

previous round, i.e., Line 3 of Alg. 3; 2) whether the public hash value  $\text{Hash}(|\mathbf{y}|_G^{(i+1)}, s_H)$  truly hashes the output of this round ( $|\mathbf{y}|_G^{(i+1)}$ ), i.e., Line 8 of Alg. 3.

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

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 in the supplementary file.

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  $\mathbf{y}$  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 in the supplementary file.

### C. Security Analysis

We claim that Alg. 1, 2, 3, and Alg.4 and 5 in the supplementary file all generate correctness proofs for the watermark detection process without exposing the  $sk$ , and the probability of using a wrong  $sk^*$  to generate a proof that can deceive the verifier is negligible. This is guaranteed by the completeness, soundness and zero-knowledge property of zero-knowledge arguments, which have been proved in previous work [15], [32], [33]. We provide the security proof for Alg. 1 in the supplementary file, and the proofs for the other four algorithms are similar. Moreover, the only viable attack, namely owners denying harmful text generation by using an inconsistent key, can be prevented by the key commitment in the Setup algorithm, where the commitment is implemented in practice using a cryptographic hash function.

## VIII. 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,

TABLE III: 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

for *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 (see the supplementary file)? 3) How efficient is *PVMark*?

#### A. Implementation Details and Setup

For the uniformity testing of hash functions, we implement SHA256, Keccak256, BLAKE2, MiMC, Poseidon, and Poseidon2 in Rust.

For the proof construction of watermark detection, we use circom [34], a domain-specific language for defining arithmetic circuits, to construct constraints based on *circomlib* [35]. 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 [36], 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 [37], which can compile circom circuits to Nova [38]. All variants of the implemented watermark detection are shown in Table III.

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 [39], widely used in blockchain project such as Ethereum [40] and Algorand [41]. BN254 offers faster computation due to its smaller field size compared to curves like BLS12-381.

**Parameter settings.** For watermark embedding and detection, we follow the settings of the original work [3], [4], [12]. For KGW, we set the green list size  $\gamma = 0.25$  and the recent context window width  $\psi = 1$ . For SynthID-Text, we set the recent 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 size  $\gamma = 0.5$ , the recent context window width  $\psi = 1$ , the total number of positions  $\hat{n} = 6$  in *msg* and the bit length  $\hat{m} = 4$  per position.

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.

#### B. 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 use the avalanche effect coefficient and chi-square test to evaluate the randomness and uniformity of hash functions. The avalanche effect is a desirable property of cryptographic hash functions, suggesting an average of one-half of the output bits should change whenever a single input bit is flipped [42]. A coefficient closer to 0.5 indicates better adherence to the avalanche effect property. We compute the avalanche effect coefficient by [43]. Let  $N_1, N_2$  denote the number of input bits and output bits, respectively. We set  $N_1 = N_2 = 256$  for SHA256, Keccak256, BLAKE2 hash and  $N_1 = N_2 = 254$  for MiMC, Poseidon, Poseidon2 hash, and execute the algorithm 5,000,000 times to take the average. 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  $\chi^2$  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  $\chi^2$  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 IV. The average and standard deviation of multiple chi-squared tests statistic  $\chi^2$ , along with the success rate of passing the tests, are shown at the bottom of Table IV. 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*. Due to space constraint, the effectiveness, fidelity and robustness results of the overall watermarking scheme are moved to the supplementary file. The conclusion is that the effectiveness, fidelity and robustness of watermarking is almost the same with or without *PVMark*, which leaves the efficiency of *PVMark* to be questioned.

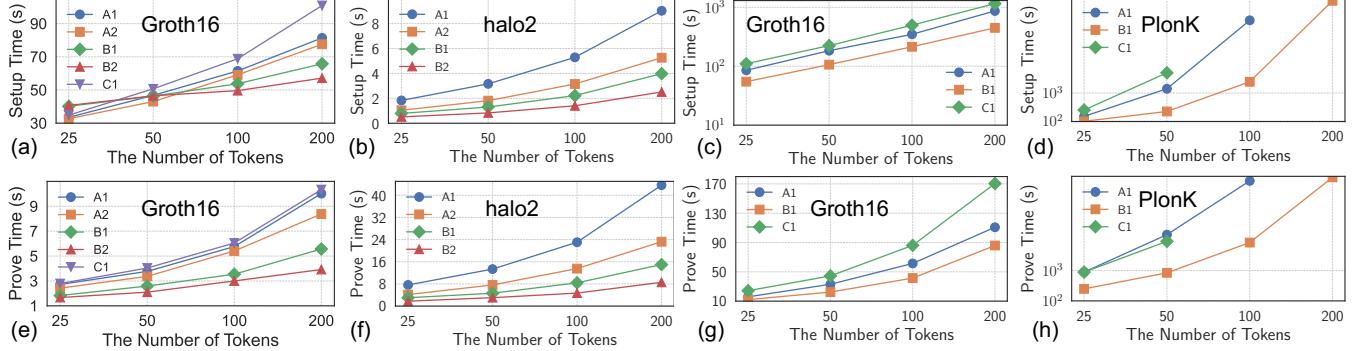
#### C. Efficiency

##### Proof generation for zero-bit watermarking schemes.

Fig. 2 and Fig. 3 present the ZKP costs for various *PVMark* variants using four different ZKP protocols: Groth16, halo2, Plonk and Nova. ZKP costs include the setup time, prove time, verification time, and proof size. We do not present the verification time, as the slowest (PlonK) verification time does

TABLE IV: 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
$\chi^2 \pm s$	$370.322 \pm 40.496$	$372.057 \pm 40.068$	$370.755 \pm 40.157$	$10.309 \pm 5.321$	$10.218 \pm 5.180$	$10.209 \pm 5.175$
Success Rate (%)	0	0	0	99.1	99.2	99.4

Fig. 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.TABLE V: 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

TABLE VI: 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-Watermark	1/1/1	2/2/2	2/2/2	4/4/2	5/5/1	5/8/5

not exceed two seconds, which is far less than the setup and prove time.

In Fig. 2, it is found that although the setup time of Groth16 is higher than that of halo2, their prove time shows the opposite. But still, the total time of halo2 is 1-5× faster than Groth16, mostly due to our careful design of circuit chips and the lookup table employed. PlonK, unfortunately, delivers exorbitant setup time and prove time, as its constraints are compiled by the universal Circom compiler, and restricted by the high-level programming language circom, preventing us from tailoring the circuit to the application. Among the variants of hash function, the three-to-one hash effectively reduces the cost compared to the two-to-one hash, and the saving grows with more tokens to be verified. Among all hash variants, Poseidon hash has the lowest overhead, exhibiting its suitability 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 V. Hence we consider the running time is reasonable in a watermark verification setting.

Table VII 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

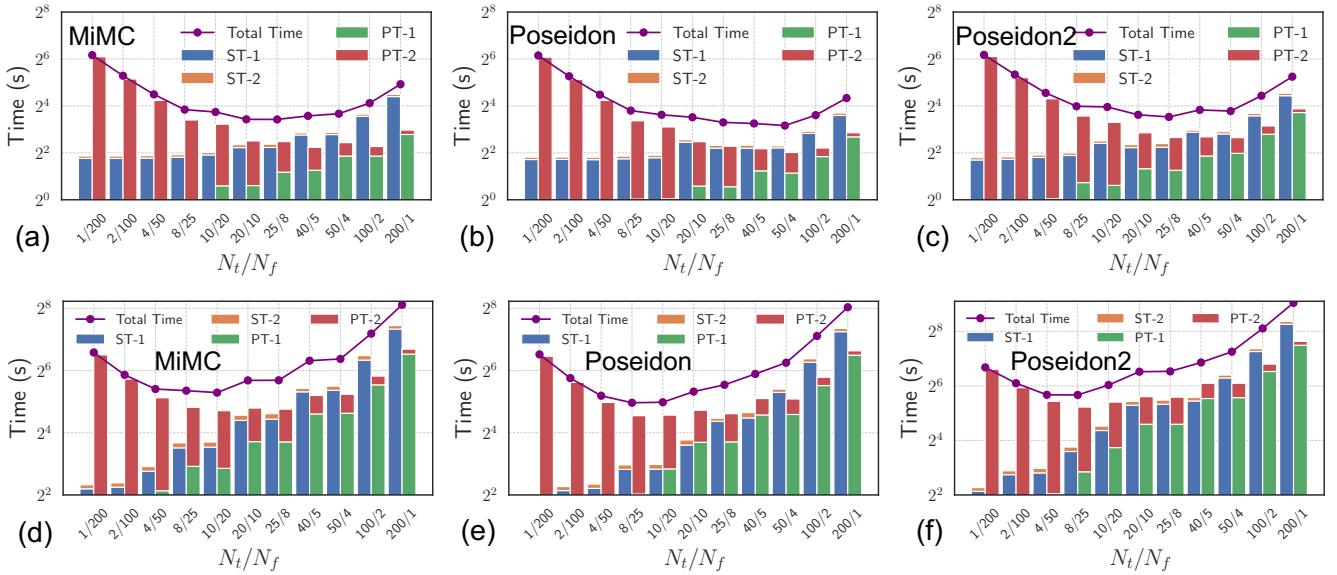


Fig. 3: ZKP costs for Nova version of *PVMark* v.s. varying  $N_t, N_f$  ( $N_t \times N_f = 200$  tokens): the upper row is for detection by KGW and the lower row is for detection by SynthID-Text. ST-1, ST-2 denote the setup time to prove the final instance and the folding process, respectively, and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

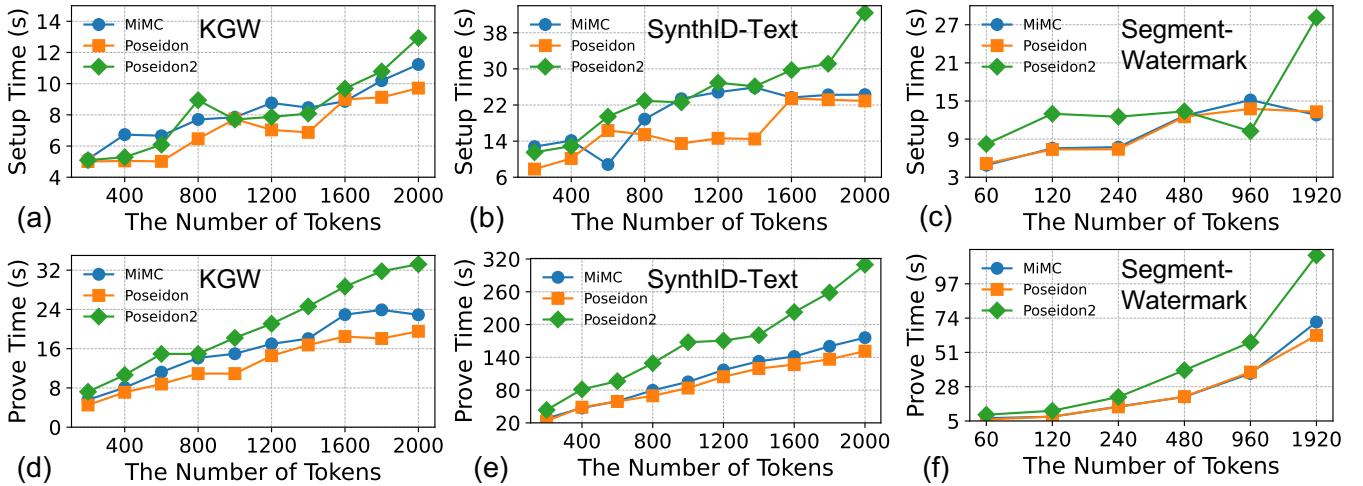


Fig. 4: ZKP costs for Nova version of *PVMark* v.s. different numbers of tokens: the setting of  $N_t$  is such that the sum of setup time and prove time is minimized.

TABLE VII: 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		Poseidon2
	2to1	3to1	2to1	3to1	
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

utilizes Spartan [44] 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 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 VI.

## IX. 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.

## REFERENCES

- [1] Y. Pan, L. Pan, W. Chen, P. Nakov, M.-Y. Kan, and W. Y. Wang, "On the risk of misinformation pollution with large language models," 2023. [Online]. Available: <https://arxiv.org/abs/2305.13661>
- [2] C. Vasilatos, M. Alam, T. Rahwan, Y. Zaki, and M. Maniatakos, "Howkgpt: Investigating the detection of chatgpt-generated university student homework through context-aware perplexity analysis," 2023. [Online]. Available: <https://arxiv.org/abs/2305.18226>
- [3] J. Kirchenbauer, J. Geiping, Y. Wen, J. Katz, I. Miers, and T. Goldstein, "A watermark for large language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 17061–17084.
- [4] S. Dathathri, A. See, S. Ghaisas, P.-S. Huang, R. McAdam, J. Welbl, V. Bachani, A. Kaskasoli, R. Stanforth, T. Matejovicova *et al.*, "Scalable watermarking for identifying large language model outputs," *Nature*, vol. 634, no. 8035, pp. 818–823, 2024.
- [5] T. Lee, S. Hong, J. Ahn, I. Hong, H. Lee, S. Yun, J. Shin, and G. Kim, "Who wrote this code? watermarking for code generation," 2024. [Online]. Available: <https://arxiv.org/abs/2305.15060>
- [6] L. Wang, W. Yang, D. Chen, H. Zhou, Y. Lin, F. Meng, J. Zhou, and X. Sun, "Towards codable watermarking for injecting multi-bits information to llms," 2024. [Online]. Available: <https://arxiv.org/abs/2307.15992>
- [7] K. Yoo, W. Ahn, J. Jang, and N. Kwak, "Robust multi-bit natural language watermarking through invariant features," in *ACL*, 2023, pp. 2092–2115.
- [8] J. Kirchenbauer, J. Geiping, Y. Wen, M. Shu, K. Saifullah, K. Kong, K. Fernando, A. Saha, M. Goldblum, and T. Goldstein, "On the reliability of watermarks for large language models," 2024. [Online]. Available: <https://arxiv.org/abs/2306.04634>
- [9] X. Zhao, P. Ananth, L. Li, and Y.-X. Wang, "Provable robust watermarking for ai-generated text," 2023. [Online]. Available: <https://arxiv.org/abs/2306.17439>
- [10] Z. Hu, L. Chen, X. Wu, Y. Wu, H. Zhang, and H. Huang, "Unbiased watermark for large language models," 2023. [Online]. Available: <https://arxiv.org/abs/2310.10669>
- [11] Y. Wu, Z. Hu, J. Guo, H. Zhang, and H. Huang, "A resilient and accessible distribution-preserving watermark for large language models," 2024. [Online]. Available: <https://arxiv.org/abs/2310.07710>
- [12] W. Qu, W. Zheng, T. Tao, D. Yin, Y. Jiang, Z. Tian, W. Zou, J. Jia, and J. Zhang, "Provably robust multi-bit watermarking for ai-generated text," 2025. [Online]. Available: <https://arxiv.org/abs/2401.16820>
- [13] J. Fairuze, S. Garg, S. Jha, S. Mahloujifar, M. Mahmoodi, and M. Wang, "Publicly-detectable watermarking for language models," 2024. [Online]. Available: <https://arxiv.org/abs/2310.18491>
- [14] A. Liu, L. Pan, X. Hu, S. Li, L. Wen, I. King, and S. Y. Philip, "An unforgeable publicly verifiable watermark for large language models," in *The Twelfth International Conference on Learning Representations*, 2023.
- [15] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes," in *Annual International Cryptology Conference*. Springer, 2022, pp. 359–388.
- [16] K. Krishna, Y. Song, M. Karpinska, J. Wieting, and M. Iyyer, "Paraphrasing evades detectors of ai-generated text, but retrieval is an effective defense," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [17] E. Mitchell, Y. Lee, A. Khazatsky, C. D. Manning, and C. Finn, "Detectgpt: Zero-shot machine-generated text detection using probability curvature," in *International Conference on Machine Learning*. PMLR, 2023, pp. 24950–24962.
- [18] V. Verma, E. Fleisig, N. Tomlin, and D. Klein, "Ghostbuster: Detecting text ghostwritten by large language models," in *ACL*, Jun. 2024, pp. 1702–1717.
- [19] A. Hans, A. Schwarzschild, V. Cherepanova, H. Kazemi, A. Saha, M. Goldblum, J. Geiping, and T. Goldstein, "Spotting llms with binoculars: Zero-shot detection of machine-generated text," 2024. [Online]. Available: <https://arxiv.org/abs/2401.12070>
- [20] A. M. Elkhatat, K. Elsaied, and S. Almeer, "Evaluating the efficacy of ai content detection tools in differentiating between human and ai-generated text," *International Journal for Educational Integrity*, vol. 19, no. 1, p. 17, 2023.
- [21] X. Zhao, P. V. Ananth, L. Li, and Y.-X. Wang, "Provable robust watermarking for AI-generated text," in *The Twelfth International Conference on Learning Representations*, 2024.
- [22] Q. Pang, S. Hu, W. Zheng, and V. Smith, "No free lunch in llm watermarking: Trade-offs in watermarking design choices," 2024. [Online]. Available: <https://arxiv.org/abs/2402.16187>
- [23] E. Barker and J. Kelsey, "Recommendation for random number generation using deterministic random bit generators," 2015-06-24 2015.
- [24] R. C. Tausworthe, "Random numbers generated by linear recurrence modulo two," *Mathematics of Computation*, vol. 19, no. 90, pp. 201–209, 1965.
- [25] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM Journal on computing*, vol. 15, no. 2, pp. 364–383, 1986.
- [26] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.
- [27] G. Marsaglia, "Xorshift rngs," *Journal of Statistical software*, vol. 8, pp. 1–6, 2003.
- [28] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 191–219.
- [29] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schafnecker, "Poseidon: A new hash function for {Zero-Knowledge} proof systems," in *30th USENIX Security Symposium*, 2021, pp. 519–535.
- [30] L. Grassi, D. Khovratovich, and M. Schafnecker, "Poseidon2: A faster version of the poseidon hash function," in *International Conference on Cryptology in Africa*. Springer, 2023, pp. 177–203.
- [31] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Waldfish, "Taking proof-based verified computation a few steps closer to practicality," in *21st USENIX Security Symposium*, 2012, pp. 253–268.
- [32] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology-EUROCRYPT*. Springer, 2016, pp. 305–326.
- [33] A. Gabizon, Z. J. Williamson, and O. Ciobanu, "PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019. [Online]. Available: <https://eprint.iacr.org/2019/953>
- [34] iden3, "Circos," <https://github.com/iden3/circos>, 2021.
- [35] ———, "circomlib," <https://github.com/iden3/circomlib>, 2021.
- [36] Zcash, "halo2," <https://github.com/zcash/halo2>, 2022.
- [37] nalinbhardwaj, "Nova-scotia," <https://github.com/nalinbhardwaj/Nova-Scotia>, 2023.
- [38] microsoft, "Nova," <https://github.com/microsoft/Nova>, 2022.
- [39] T. Kim and R. Barbulescu, "Extended tower number field sieve: A new complexity for the medium prime case," in *Annual international cryptology conference*. Springer, 2016, pp. 543–571.
- [40] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [41] J. Chen and S. Micali, "Algorand," 2017. [Online]. Available: <https://arxiv.org/abs/1607.01341>
- [42] A. F. Webster and S. E. Tavares, "On the design of s-boxes," in *Conference on the theory and application of cryptographic techniques*. Springer, 1985, pp. 523–534.
- [43] A. Sateesan. (2020) Analyze your hash functions: The avalanche metrics calculation. [Online]. Available: <https://arishs.medium.com/analyze-your-hash-functions-the-avalanche-metrics-calculation-767b7445ee6f>
- [44] S. Setty, "Spartan: Efficient and general-purpose zksnarks without trusted setup," in *Annual International Cryptology Conference*. Springer, 2020, pp. 704–737.