# Epistle: Elastic Succinct Arguments for Plonk Constraint System

Shuangjun Zhang, Fudan University
Dongliang Cai, Fudan University
Yuan Li, Fudan University
Haibin Kan, Fudan University
Liang Zhang, Hainan University

*Date: June 1, 2024*

**Abstract**

We study elastic SNARKs, a concept introduced by the elegant work of Gemini (EUROCRYPTO 2022). The prover of elastic SNARKs has multiple configurations with different time and memory tradeoffs and the output proof is independent of the chosen configuration. In addition, during the execution of the protocol, the space-efficient prover can pause the protocol and save the current state. The time-efficient prover can then resume the protocol from that state. Gemini constructs an elastic SNARK for R1CS.

We present Epistle, an elastic SNARK for Plonk constraint system. For an instance with size $N$, in the time-efficient configuration, the prover uses $O_\lambda(N)$ cryptographic operations and $O(N)$ memory; in the space-efficient configuration, the prover uses $O_\lambda(N \log N)$ cryptographic operations and $O(\log N)$ memory. Compared to Gemini, our approach reduces the asymptotic time complexity of the space-efficient prover by a factor of $\log N$. The key technique we use is to make the toolbox for multivariate PIOP provided by HyperPlonk (EUROCRYPTO 2023) elastic, with the most important aspect being the redesign of each protocol in the toolbox in the streaming model.

We implement Epistle in Rust. Our benchmarks show that Epistle maintains a stable memory overhead of around 1.5 GB for instance sizes exceeding $2^{21}$, while the time overhead shows a linear growth trend.

**Keywords:** SNARKs, Streaming algorithms

## Contents

## 1 Introduction

*Succinct non-interactive arguments of knowledge* (SNARKs) [1–4] are powerful cryptographic tools for checking some statements holding with succinct proofs and fast verifications. In recent years, research on SNARKs has received extensive attention in both theoretical and engineering fields. For example, SNARKs provide solutions for blockchain scalability. SNARKs can rapidly verify whether a large set of transactions is valid by proving the correctness of a certain computation (program) in a short amount of time. In fact, SNARKs are widely used in verifiable computation. SNARKs can generate succinct (possibly zero-knowledge) proo for long computation, and a weak verifier can efficiently (in logarithmic time relative to the original computation) verify whether the computation is correct. However, designing efficient SNARKs, especially reducing the prover time/space complexity, presents an intricate challenge.

Plonk, invented by Gabizon et al. [5], is currently one of the most popular SNARKs. The overall strategy of Plonk is to convert a program to an arithmetic circuit, commonly referred to as *Plonkish* circuits, and then verify the correctness of the arithmetic circuit. For an arithmetic circuit with $N$ gates, the prover runs in time $O_\lambda(N \log N)$[1] to generate a proof with size $O(1)$ and the verifier runs in time $O_\lambda(1)$ to verify the validity of the proof. The main bottleneck for Plonk's prover lies in the need to perform *Fast Fourier Transforms* (FFT). HyperPlonk [6] replaces the FFT (used in Plonk) by the classical SumCheck protocol at its core, resulting in linear proof generation time. However, both Plonk and HyperPlonk consume a lot of memory in proof generation. Experiments in Pianist [7] show that for a circuit of size $2^{25}$, Plonk's prover requires 200GB of memory for the proof generation. Pianist then proposes a fully distributed SNARK through distributed proof generation across multiple machines with minimal communication among machines.

Gemini [8] adopts streaming algorithms to address the aforementioned memory overhead issue. The term *streaming algorithm*, loosely speaking, refers to an algorithm that does not load all input into memory at once but rather processes it one at a time in a certain (fixed) order. Gemini achieves logarithmic space complexity with quasilinear prover time. Furthermore, their construction is "elastic", which means that the prover has multiple configurations with different time and memory tradeoffs and the output proof is independent of the chosen configuration. They construct an elastic SNARK for *rank-1 constraint system* (R1CS), an $\mathcal{NP}$-complete language; roughly speaking, an R1CS instance is a vector equation that can capture the arithmetic circuit computation (slightly different from Plonkish circuits). For an R1CS instance with size $N$ (where $N$ corresponds to the number of gates in an arithmetic circuit), their construction supports two different configurations for prover:

- a *time-efficient* prover that runs in $O_\lambda(N)$ time and $O(N)$ space,
- a *space-efficient* prover that runs in $O_\lambda(N \log^2 N)$ time and $O(\log N)$ space.

Regardless of the prover configuration, the verification time is $O_\lambda(|\mathbf{x}| + \log N)$ and proof size is $O(\log N)$, where $\mathbf{x}$ is the public input of the circuit.

[1]For time complexity, we use the standard big-$O$ notation $O(\cdot)$ to denote the number of field operations and use $O_\lambda(\cdot)$ to denote the number of cryptographic operations, where $\lambda$ is the security parameter. Formally, $O_\lambda(T) = O(\lambda^c T)$ for some constant $c$.

**Table 1:** Comparison with the most relevant SNARKs

| | Prover's configuration | Prover time | Prover space | Proof size | Verifier time |
|---|---|---|---|---|---|
| Plonk | Time-efficient prover | $O_\lambda(N \log N)$ | $O(N)$ | $O(1)$ | $O_\lambda(\lvert\mathbf{x}\rvert)$ |
| HyperPlonk | Time-efficient prover | $O_\lambda(N)$ | $O(N)$ | $O(\log N)$ | $O_\lambda(\lvert\mathbf{x}\rvert + \log N)$ |
| Gemini | Time-efficient prover | $O_\lambda(N)$ | $O(N)$ | $O(\log N)$ | $O_\lambda(\lvert\mathbf{x}\rvert + \log N)$ |
| | Space-efficient prover | $O_\lambda(N \log^2 N)$ | $O(\log N)$ | $O(\log N)$ | $O_\lambda(\lvert\mathbf{x}\rvert + \log N)$ |
| Our scheme | Time-efficient prover | $O_\lambda(N)$ | $O(N)$ | $O(\log N)$ | $O_\lambda(\lvert\mathbf{x}\rvert + \log N)$ |
| | Space-efficient prover | $O_\lambda(N \log N)$ | $O(\log N)$ | $O(\log N)$ | $O_\lambda(\lvert\mathbf{x}\rvert + \log N)$ |

## 1.1 Our Contribution

We continue the exploration of elastic SNARKs. Drawing inspiration from the elegant work of Gemini, we integrate techniques from Gemini into HyperPlonk. We present Epistle, an elastic SNARK for Plonk constraint system. Our contribution shows that for proof generation time, the space-efficient prover of Epistle can save a logarithmic overhead compared to Gemini.

**Theorem 1.1.** *There exists an elastic SNARK for Plonk constraint system with two different configurations for prover:*

- *a time-efficient prover that runs in $O_\lambda(N)$ time and $O(N)$ space,*
- *a space-efficient prover that runs in $O_\lambda(N \log N)$ time and $O(\log N)$ space.*

*The verification time is $O_\lambda(|\mathbf{x}| + \log N)$ and proof size is $O(\log N)$, where $N$ denotes the number of gates of the circuit and $\mathbf{x}$ is the public input of the circuit. In addition, during the execution of the protocol, the space-efficient prover can pause the protocol and save the current state and the time-efficient prover can then resume the protocol from that state.*

Similar to the construction of most modern SNARKs, our SNARK relies on two components: polynomial IOP [9–11] and polynomial commitment [12]. Finally, the protocol is transformed into a non-interactive form through the standard *Fiat-Shamir transformation* [13]. Our starting point is HyperPlonk, a linear time SNARK for the Plonkish circuit. HyperPlonk provides a toolbox for multivariate polynomials and we observe that most polynomial IOPs in the toolbox (except for the lookup protocol) can be realized elastically. By employing techniques from Gemini, we provide a toolbox for multivariate polynomials with elastic prover and we construct an elastic polynomial IOP for Plonkish circuit based on this toolbox. For the polynomial commitment part, Gemini adopts univariate KZG polynomial commitment [12] in their construction since it can be realized elastically. We work with multilinear polynomials since HyperPlonk does, we find that the multilinear KZG scheme [14] can also be realized elastically. In Table 1, we compare our work with the most relevant SNARKs. Note that Plonk and HyperPlonk did not consider space complexity, so their prover's configuration is time-efficient only.

## 1.2 Discussions

**Degree of custom gates.** In our work, for simplicity, we treat the max degree of custom gates as a constant. For instance, in Scroll [15], the maximum degree of custom gates does not exceed 9.

**The lookup protocol.** The lookup protocol [16] is used to prove that all elements in a vector $\mathbf{f}$ are contained in another vector $\mathbf{g}$. Plonk uses the lookup protocol to support lookup gates. The most crucial step in the lookup protocol is to construct a vector $\mathbf{w} = \mathrm{sort}(\mathbf{f}, \mathbf{g})$. Here, $\mathrm{sort}(\mathbf{f}, \mathbf{g})$ means concatenating the vectors $\mathbf{f}$ and $\mathbf{g}$, and then sorting the concatenated vector according to the order of elements in $\mathbf{g}$. However, generating the stream $\mathbf{w}$ using limited space in the streaming model is challenging, so we do not design the lookup protocol in the streaming model. Gemini uses the lookup protocol to achieve holography. In their protocol, the order of elements in vector $\mathbf{f}$ matches the order in $\mathbf{g}$, allowing them to construct the stream $\mathbf{w}$ using a merge sort approach. We leave the construction of the lookup protocol for the general case in the streaming model as future work.

**Error correcting code in the streaming model.** HyperPlonk adopts a variant of Orion [17] as its polynomial commitment scheme, which relies on linear-time encodable codes [18]. However, constructing good linear codes with encoding algorithms featuring low space complexity poses a significant challenge, and there are some negative results [19, 20] in this domain.

**Customizable constraint systems.** Setty et al. [21] introduced a new generalization of R1CS that they called *Customizable Constraint System (CCS)*. They showed that CCS simultaneously generalizes Plonkish, AIR (algebraic intermediate representation) [22], and R1CS without overhead. They observed that known polynomial IOPs for R1CS (Spartan [23] and Marlin [10]) extend easily to handle CCS. They referred to these generalizations as SuperSpartan and SuperMarlin respectively. As a corollary, SuperSpartan provides a linear-time polynomial IOP for Plonkish, similar to HyperPlonk, but via a different route. Our work shows that in the streaming model, if we convert Plonkish to CCS and employ a construction similar to Gemini, we incur additional logarithmic overhead for proof generation time. We have explained why this overhead arises in the previous section since the techniques used to handle CCS are nearly identical to those used for R1CS.

## 1.3 Additional Related Works

The first construction of SNARK [1, 2] was provided by the famous PCP theorem [24, 25]. Many techniques employed in modern SNARKs have their roots in the PCP theorem, including SumCheck protocol and low-degree testing.

**Time-efficient SNARKs.** However, The concrete efficiency of PCPs is not acceptable for practice. As demonstrated in Pinocchio [26], verifying small instances would take hundreds to trillions of years. Eli Ben-Sasson et al. introduced the concept of Interactive Oracle Proof (IOP) [9], which can be viewed as an *interactive version* of PCP. Benedikt Bünz et al. introduced a variant of IOP called polynomial IOP [11], where the messages sent by the prover can be a polynomial oracle, and the verifier can query the values of this polynomial at any point. SNARK can be constructed from polynomial IOP and polynomial commitment [12], which is a cryptographic primitive designed for efficient verification of polynomial evaluations. Recently, there has been a series of exciting work focused on reducing the time complexity of the prover [6, 17, 27–

30]. The overall strategy of the aforementioned works is to replace the FFT, which takes $O(N \log N)$ time by the classical SumCheck protocol and use a linear time encodable error correcting code when designing polynomial commitment scheme. None of the protocols cited in this paragraph considers space complexity.

**Complexity-preserving SNARKs.** There is also a line of works [31–34] that consider not only the time complexity of the prover but also its *space complexity*. Bitansky and Chiesa [31] were the first to consider optimizing both time and space complexity for SNARKs simultaneously. They introduced the notion of *complexity-preserving SNARKs*, meaning both the time and space complexity required to generate a proof (for a computation) is close to the complexity of the computation itself (up to a polylogarithmic factor). Holmgren and Rothblum [32] explored complexity-preserving SNARKs for RAM computations; assuming the RAM computation requires time $T$ and space $S$, the prover achieves time complexity $\tilde{O}(T)$ and space complexity $S+o(S)$. Holmgren and Rothblum's construction relies on complex PCP and Fully Homomorphic Encryption (FHE). Block et al. [33, 34] further optimized the scheme proposed by Holmgren and Rothblum, building on modern constructions, namely polynomial IOP and polynomial commitment. They provided space-efficient polynomial commitment schemes in the streaming model based on Bulletproof [35] and DARK [11]. The ultimate goal of complexity-preserving SNARKs is to generate proofs in $O(T)$ time and $O(S)$ space, but the currently known protocols are still far from this goal. Our work slightly deviates from this line; we consider the computation of arithmetic circuits rather than RAM computations. We measure the complexity of arithmetic circuits by the number of gates in the circuit, using it as the size of our input instances.

### 1.4 Organization of the Paper

We first provide a technical overview of our scheme in Section 2. Section 3 presents the preliminaries used in this paper. In Section 4, we provide a toolbox for multivariate polynomials with elastic prover. In Section 5, we demonstrate how to use the toolbox from Section 4 to construct an elastic polynomial IOP for Plonk constraint system. We present the elastic multilinear KZG scheme in Section 6. Finally, in Section 7, we present our experimental results.

## 2 Technical Overview

As mentioned before, our starting point is HyperPlonk. Our elastic SNARK is essentially the same as HyperPlonk in time-efficient mode. In this section we give a high-level overview of how to design a space-efficient prover for HyperPlonk. We first review the basic framework of HyperPlonk.

**A review of HyperPlonk.** Let $C$ be an arithmetic circuit with a total of $N = 2^n$ gates, where each gate has fan-in two and can be one of addition, multiplication, input or a custom gate $G : \mathbb{F}^2 \to \mathbb{F}$. Let $\mathbf{x} \in \mathbb{F}^{N_{in}}$ ($N_{in} = 2^{n_{in}}$ and $N_{in} \leq N$) be a public input to the circuit. Each gate is indexed by an element $\mathbf{v} \in \{0,1\}^n$ and each input gate is indexd by an element from $0^{n-n_{in}} \times \{0,1\}^{n_{in}}$ (i.e., the first $N_{in}$ gates out of $N$ gates). HyperPlonk uses three vectors $(\mathbf{l}, \mathbf{r}, \mathbf{o}) \in \left(\mathbb{F}^N\right)^3$ to represent the computation trace of a circuit, where $\mathbf{l}$ represents the left input of each gate, $\mathbf{r}$ represents the right input, and $\mathbf{o}$ represents the output. We assume that the elements in a vector $\mathbf{f}$ with length $N = 2^n$ are indexed by $\mathbf{v} \in \{0,1\}^n$, and we denote the $\mathbf{v}$-th element of $\mathbf{f}$ as $\mathbf{f}(\mathbf{v})$. The prover needs to prove two things to the verifier.

- **The gate identity:** Let $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3) \in \left(\mathbb{F}^N\right)^3$ be three selector vectors. To prove that all gate constraints are correct, the prover needs to prove to the verifier that for all $\mathbf{v} \in \{0,1\}^n$:

$$\mathbf{s}_1(\mathbf{v}) \cdot (\mathbf{l}(\mathbf{v}) + \mathbf{r}(\mathbf{v})) + \mathbf{s}_2(\mathbf{v}) \cdot \mathbf{l}(\mathbf{v}) \cdot \mathbf{r}(\mathbf{v}) + \mathbf{s}_3(\mathbf{v}) \cdot G\,(\mathbf{l}(\mathbf{v}), \mathbf{r}(\mathbf{v})) - \mathbf{o}(\mathbf{v}) + \mathbf{x}'(\mathbf{v}) = 0, \tag{1}$$

  where $\mathbf{x}' = \mathbf{x}||\mathbf{0} \in \mathbf{F}^N$ (padding $\mathbf{x}$ with $N - N_{in}$ zeros). The three selector vectors are defined as follows:
  - for an addition gate: $\mathbf{s}_1(\mathbf{v}) = 1, \mathbf{s}_2(\mathbf{v}) = 0, \mathbf{s}_3(\mathbf{v}) = 0$,
  - for a multiplication gate: $\mathbf{s}_1(\mathbf{v}) = 0, \mathbf{s}_2(\mathbf{v}) = 1, \mathbf{s}_3(\mathbf{v}) = 0$,
  - for a custom gate: $\mathbf{s}_1(\mathbf{v}) = 0, \mathbf{s}_2(\mathbf{v}) = 0, \mathbf{s}_3(\mathbf{v}) = 1$,
  - for an input gate: $\mathbf{s}_1(\mathbf{v}) = 0, \mathbf{s}_2(\mathbf{v}) = 0, \mathbf{s}_3(\mathbf{v}) = 0$.

  It is easy to verify that the definitions of these three vectors are "appropriate". For any vector $\mathbf{f} \in \mathbb{F}^N$ with length $N = 2^n$, we can view it as a function $\mathbf{f} : \{0,1\}^n \to \mathbb{F}$, let $\widehat{\mathbf{f}}(X_1, \cdots, X_n) \in \mathbb{F}[X_1, \cdots, X_n]$ be its *multilinear extension* (see Section 3.3) and define a polynomial $P \in \mathbb{F}[X_1, \cdots, X_n]$:

$$P(\mathbf{X}) = \widehat{\mathbf{s}}_1(\mathbf{X}) \cdot \left(\widehat{\mathbf{l}}(\mathbf{X}) + \widehat{\mathbf{r}}(\mathbf{X})\right) + \widehat{\mathbf{s}}_2(\mathbf{X}) \cdot \widehat{\mathbf{l}}(\mathbf{X}) \cdot \widehat{\mathbf{r}}(\mathbf{X}) + \widehat{\mathbf{s}}_3(\mathbf{X}) \cdot G\left(\widehat{\mathbf{l}}(\mathbf{X}), \widehat{\mathbf{r}}(\mathbf{X})\right) - \widehat{\mathbf{o}}(\mathbf{X}) + \widehat{\mathbf{x}}'(\mathbf{X}), \tag{2}$$

  where $\mathbf{X} = (X_1, \cdots, X_n)$. Then for checking the gate identity constraints, it is suffice to check that $P(\mathbf{v}) = 0$ for all $\mathbf{v} \in \{0,1\}^n$. This is achieved by ZeroCheck, which is based on the classical SumCheck protocol [36].
- **The wiring identity:** Let $\mathbf{w} = \mathbf{l}||\mathbf{r}||\mathbf{o}||\mathbf{0} \in \mathbb{F}^{4N}$ (we pad $\mathbf{w}$ such that the length of $\mathbf{w}$ is a power of two), then in this vector, there are some equal values (for example, the output of the 5-th gate is the left input of the 8-th gate). In HyperPlonk, the wiring identity constraints are captured by a permutation $\boldsymbol{\tau} : \{0,1\}^{2n} \to \{0,1\}^{2n}$. The prover needs to prove the verifier that for all $\mathbf{v}' \in \{0,1\}^{2n}$, $\mathbf{w}(\mathbf{v}') = \mathbf{w}(\boldsymbol{\tau}(\mathbf{v}'))$. This is achieved by a prescribed permutation check protocol, which is based on the product check protocol.

**The streaming model.** To construct a space-efficient SNARK, we cannot load all the vectors from the HyperPlonk instance into memory at once (which would require $O(N)$ memory). Instead, we assume the prover accesses the vectors in a streaming fashion. Suppose $\mathbf{f} \in \mathbb{F}^N$, in the *streaming model*, a streaming algorithm can only scan the vector in the given order. We use the notation $\mathcal{S}(\mathbf{f})$ to denote the stream of $\mathbf{f}$. A streaming algorithm can perform two operations on a stream: **init** and **next**. The **next** operation returns the next element in the stream, while the **init** operation resets the stream to its initial state, i.e., back to the first element of the stream. The streaming algorithm does *not* allow random access to elements of $\mathbf{f}$.

**SumCheck in the streaming model.** In HyperPlonk, the foundational protocol is the classical SumCheck protocol. We use techniques from Gemini, which is also borrowed from [37] to implement a space-efficient SumCheck in the streaming model. Recall that the goal of SumCheck protocol is for the prover to convince the verifier that $\sum_{\mathbf{v} \in \{0,1\}^n} P(\mathbf{v}) = \gamma$ for a multivariate polynomial $P(X_1, \cdots, X_n) \in \mathbb{F}[X_1, \cdots, X_n]$ and a field element $\gamma \in \mathbb{F}$. Let $\mathbf{f} \in \mathbb{F}^N$ for some $N = 2^n$, we can view $\mathbf{f}$ as a function $\mathbf{f} : \{0,1\}^n \to \mathbf{F}$. Let $\widehat{\mathbf{f}} \in \mathbb{F}[X_1, \cdots, X_n]$ be the multilinear extension of $\mathbf{f}$, we first explain how to perform a SumCheck for the multilinear polynomial $\widehat{\mathbf{f}}$. In the first round of the SumCheck protocol, the prover constructs and sends an univariate polynomial

$$\widehat{\mathbf{f}}_n(X_n) = \sum_{v_1,\cdots,v_{n-1}\in\{0,1\}} \widehat{\mathbf{f}}(v_1,\cdots,v_{n-1},X_n) \tag{3}$$

to the verifier. Since $\widehat{\mathbf{f}}$ is multilinear, it follows that for any $v_1,\cdots,v_{n-1}\in\{0,1\}$,

$$\widehat{\mathbf{f}}(v_1,\cdots,v_{n-1},X_n) = \mathbf{f}(v_1,\cdots,v_{n-1},0)\cdot(1-X_n) + \mathbf{f}(v_1,\cdots,v_{n-1},1)\cdot X_n, \tag{4}$$

which can be computed from two adjacent elements in vector $\mathbf{f}$. Therefore, the univariate polynomial $\widehat{\mathbf{f}}_n(X_n)$ can be computed by making one pass over the stream $\mathcal{S}(\mathbf{f})$ with $O(N)$ field operations. After receiving $\widehat{\mathbf{f}}_n(X_n)$, the verifier first checks that $\widehat{\mathbf{f}}_n(0)+\widehat{\mathbf{f}}_n(1)=\gamma$, then samples a random value $r_n \leftarrow \mathbb{F}$, and then use the SumCheck recursively to check that

$$\widehat{\mathbf{f}}_n(r_n) = \sum_{v_1,\cdots,v_{n-1}\in\{0,1\}} \widehat{\mathbf{f}}(v_1,\cdots,v_{n-1},r_n). \tag{5}$$

Let $\mathbf{f}^{(1)} \in \mathbb{F}^{N/2}$ be the vector such that for all $\mathbf{v}=(v_1,\cdots,v_{n-1})\in\{0,1\}^{n-1}$,

$$\mathbf{f}^{(1)}(\mathbf{v}) = \widehat{\mathbf{f}}(v_1,\cdots,v_{n-1},r_n) = \mathbf{f}(v_1,\cdots,v_{n-1},0)\cdot(1-r_n) + \mathbf{f}(v_1,\cdots,v_{n-1},1)\cdot r_n, \tag{6}$$

then the stream $\mathcal{S}\left(\mathbf{f}^{(1)}\right)$ can be constructed from $\mathcal{S}(\mathbf{f})$ and one pass over $\mathcal{S}\left(\mathbf{f}^{(1)}\right)$ needs one pass over $\mathcal{S}(\mathbf{f})$ with $O(N)$ field operations. In general, Let $\mathbf{f}^{(l)}\in\mathbb{F}^{N/2^l}$ be the vector such that for all $\mathbf{v}=(v_1,\cdots,v_{n-l})\in\{0,1\}^{n-l}$,

$$\mathbf{f}^{(l)}(\mathbf{v}) = \mathbf{f}^{(l-1)}(v_1,\cdots,v_{n-l},0)\cdot(1-r_{n-l+1}) + \mathbf{f}^{(l-1)}(v_1,\cdots,v_{n-l},1)\cdot r_{n-l+1}, \tag{7}$$

where $r_{n-l+1}$ is the random challenge value sampled and sent by the verifier. The stream $\mathcal{S}\left(\mathbf{f}^{(l)}\right)$ can be constructed from $\mathcal{S}\left(\mathbf{f}^{(l-1)}\right)$, which is based on the input stream $\mathcal{S}(\mathbf{f})$ (the prover can't store the vector $\mathbf{f}^{(l-1)}$ in the memory). It is not hard to show that one pass over $\mathcal{S}\left(\mathbf{f}^{(l)}\right)$ needs one pass over $\mathcal{S}(\mathbf{f})$ with $O(N+N/2+\cdots+N/2^{l-1})=O(N)$ field operations. Because the SumCheck protocol has a total of $n=\log N$ rounds, with each round's time complexity being $O(N)$, the overall time complexity is $O(N\log N)$ and the prover makes $O(\log N)$ passes over the input stream $\mathcal{S}(\mathbf{f})$. The space cost is $O(\log N)$ since we only need to keep track of at most two elements from each vector $\mathbf{f}^{(l)}$. Note that if we do not consider the space overhead, the prover can save the intermediate results of each round's computation (i.e., can compute $O(\mathbf{f}^{(l)})$ from vector $O(\mathbf{f}^{(l-1)})$ in $O(N/2^{l-1})$ time), so the total time complexity for the prover is $O(N+N/2+\cdots+2)=O(N)$ in the time-efficient model [38]. However, the space complexity is $O(N)$ in this model. In our work, the polynomial that need to be verified using SumCheck are not multilinear but rather a combination of multiple multilinear polynomials (for example, Equation (2)). We can use the method described above to compute multiple multilinear polynomials and then combine them accordingly. See Section 4.1 for the details of SumCheck protocol.

**Proving gate indentity constraints.** As mentioned before, for checking the gate identity constraints, it is suffice to check that $P(\mathbf{v})=0$ for all $\mathbf{v}\in\{0,1\}^n$, the polynomial $P$ is defined in Equation (2). We refer to this type of check as ZeroCheck. We cannot rely solely on running the SumCheck protocol for checking that $\sum_{\mathbf{v}\in\{0,1\}^n} P(\mathbf{v})=0$ to ensure that $P(\mathbf{v})=0$ for all $\mathbf{v}\in\{0,1\}^n$. Using the idea from Spartan, define the following polynomial

$$Q(\mathbf{X}) = \sum_{\mathbf{v}\in\{0,1\}^n} P(\mathbf{v})\cdot\widehat{\mathrm{eq}}(\mathbf{X},\mathbf{v}), \tag{8}$$

where $\widehat{\mathrm{eq}}(\mathbf{X},\mathbf{v}) = \prod_{i=1}^{n}((1-X_i)\cdot(1-v_i)+X_i\cdot v_i)$. From Schwartz-Zippel Lemma, if $P(\mathbf{v}) \neq 0$ for some $\mathbf{v} \in \{0,1\}$, then for a random vector $\boldsymbol{\rho} = (\rho_1,\cdots,\rho_n) \in \mathbb{F}^n$, the probability that $Q(\boldsymbol{\rho}) = 0$ is at most $\frac{n}{|\mathbb{F}|}$. Therefore, it is suffice to run SumCheck protocol to check that $\sum_{\mathbf{v}\in\{0,1\}^n} P(\mathbf{v})\cdot\widehat{\mathrm{eq}}(\boldsymbol{\rho},\mathbf{v}) = 0$. Let $\mathbf{T}_{\boldsymbol{\rho}} \in \mathbb{F}^N$ ($N = 2^n$) be a vector such that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{T}_{\boldsymbol{\rho}}(\mathbf{v}) = \widehat{\mathrm{eq}}(\boldsymbol{\rho},\mathbf{v})$, in the streaming model, the prover does not compute and store all elements of vector $\mathbf{T}_{\boldsymbol{\rho}}$ in memory; instead, it generates the required elements based on the explicit expression of $\widehat{\mathrm{eq}}$ when needed. In some middle rounds of the SumCheck protocol, the prover needs to construct vector $\mathbf{T}_{\boldsymbol{\rho}}^{(l)} \in \mathbb{F}^{N/2^l}$ such that for all $\mathbf{v} = (v_1,\cdots,v_{n-l}) \in \{0,1\}^{n-l}$, $\mathbf{T}_{\boldsymbol{\rho}}^{(l)}(\mathbf{v}) = \widehat{\mathrm{eq}}(\boldsymbol{\rho},v_1,\cdots,v_{n-l},r_{n-l+1},\cdots,r_n)$, where $r_{n-l+1},\cdots,r_n$ are random challenge values sampled and sent by the verifier. Unlike regular vectors, the construction of the vector $\mathbf{T}_{\boldsymbol{\rho}}^{(l)}$ does not depend on the input vector $\mathbf{T}_{\boldsymbol{\rho}}$. Instead, each element of the vector can be generated based on the expression $\widehat{\mathrm{eq}}$ with $O(\log N)$ field operations. One pass over the vector $\mathbf{T}_{\boldsymbol{\rho}}^{(l)}$ takes $O(N\log N/2^l)$ field operations. In total, the time complexity of ZeroCheck is $\sum_{l=0}^{n} O(N\log N/2^l) = O(N\log N)$ and the space complexity remains $O(\log N)$. See Section 4.2 for the details of ZeroCheck protocol.

**Product check in the streaming model.** As described earlier, we need to perform a prescribed permutation check to prove the wiring identity constraints, which is accomplished based on product check protocol. For a vector $\mathbf{f}$ with length $N = 2^n$. The main goal of product check protocol is for the prover to convince the verifier that $\prod_{\mathbf{v}\in\{0,1\}^n}\mathbf{f}(\mathbf{v}) = 1$. HyperPlonk based their construction of product check prorocol on the following theorem, which is proposed by Quark [39].

**Theorem 2.1.** $\prod_{\mathbf{v}\in\{0,1\}^n}\mathbf{f}(\mathbf{v}) = 1$ *if and only if there exists a vector* $\mathbf{g}$ *with length* $2N = 2^{n+1}$ *such that*

- $\mathbf{g}(1,\cdots,1,0) = 1$,
- *for all* $\mathbf{v} \in \{0,1\}^n$, $\mathbf{g}(0,\mathbf{v}) = \mathbf{f}(\mathbf{v})$, $\mathbf{g}(1,\mathbf{v}) = \mathbf{g}(\mathbf{v},0)\cdot\mathbf{g}(\mathbf{v},1)$.

Let $\widehat{\mathbf{g}}$ be the multilinear extension of $\mathbf{g}$, the prover sends the polynomial oracle $\widehat{\mathbf{g}}$ to the verifier. The verifier first queries $\widehat{\mathbf{g}}$ at point $(1,\cdots,1,0)$ and rejects if it is not 1, then runs the ZeroCheck protocol to ensure the second condition holds. However, in the streaming model, it is not hard to check that one pass over the stream $\mathcal{S}(\mathbf{g})$ requires $O(\log N)$ passes over the stream $\mathcal{S}(\mathbf{f})$. Since the prover of the ZeroCheck needs to make $O(\log N)$ passes over the stream $\mathcal{S}(\mathbf{g})$, the total number of passes over $\mathcal{S}(\mathbf{f})$ is $O(\log^2 N)$. This means that the time complexity of the prover is at least $O(N\log^2 N)$, which is not in line with our goal.

We use the ideas from [28] to design the product check protocol, and we find that this approach is well-suited for working in the streaming model. The prover constructs a vector $\mathbf{g} \in \mathbb{F}^N$ such that

$$\mathbf{g} = (1, \mathbf{f}(0), \mathbf{f}(0)\mathbf{f}(1), \cdots, \prod_{0\le j\le N-2}\mathbf{f}(j)),$$

where $\mathbf{f}(j)$ is the $j$-th element of $\mathbf{f}$. Let $\mathbf{g}^{\sigma}$ be the cyclic shift vector of $\mathbf{g}$, that is

$$\mathbf{g}^{\sigma} = (\mathbf{f}(0), \mathbf{f}(0)\mathbf{f}(1), \cdots, \prod_{0\le j\le N-2}\mathbf{f}(j), 1),$$

the prover sends two polynomial oracles $\widehat{\mathbf{g}}$ and $\widehat{\mathbf{g}}^{\sigma}$ to the verifier. The verifier first queries $\widehat{\mathbf{g}}$ at point $(0,\cdots,0)$ and rejects if it is not 1, then checks that $\mathbf{f}\circ\mathbf{g} = \mathbf{g}^{\sigma}$, where $\circ$ is the pair-wise product of two vectors. We can use ZeroCheck protocol to verify this relationship, as $\mathbf{f}\circ\mathbf{g} = \mathbf{g}^{\sigma}$ if and only if $\widehat{\mathbf{f}}(\mathbf{v})\cdot\widehat{\mathbf{g}}(\mathbf{v}) - \widehat{\mathbf{g}}^{\sigma}(\mathbf{v}) = 0$ for all $\mathbf{v} \in \{0,1\}^n$. More importantly, constructing streams $\mathcal{S}(\mathbf{g})$ and $\mathcal{S}(\mathbf{g}^{\sigma})$ from stream $\mathbf{f}$ is straightforward, and one pass over $\mathcal{S}(\mathbf{g})$ or $\mathcal{S}(\mathbf{g}^{\sigma})$ only requires one pass over $\mathcal{S}(\mathbf{f})$. Therefore, the time complexity of the prover is $O(N\log N)$ and the space complexity is the same as ZeroCheck protocol, which is $O(\log N)$. See Section 4.4 for the details of product check protocol, note that this protocol also requires the verifier to ensure that $\mathbf{g}^\sigma$ is a cyclic shift left vector of $\mathbf{g}$.

**Cyclic shift left check in the streaming model.** Let $\mathbf{b}$ and $\mathbf{b}^\sigma$ be two vectors with length $N = 2^n$, the main goal of cyclic shift left protocol is for the prover to convince the verifier that $\mathbf{g}^\sigma$ is a cyclic shift left vector of $\mathbf{g}$. In [28], they treat these polynomial oracles as univariate polynomial oracles, i.e., the vector $\mathbf{g}$ and $\mathbf{g}^\sigma$ corresponds to the following polynomials:

$$\begin{aligned} G(X) &= \mathbf{g}(0) + \mathbf{g}(1)X + \mathbf{g}(2)X^2 + \cdots + \mathbf{g}(N-1)X^{N-1}, \\ G^\sigma(X) &= \mathbf{g}^\sigma(0) + \mathbf{g}^\sigma(1)X + \mathbf{g}^\sigma(2)X^2 + \cdots + \mathbf{g}^\sigma(N-1)X^{N-1}. \end{aligned} \tag{9}$$

It is not hard to show that $\mathbf{g}^\sigma$ is a cyclic shift left vector of $\mathbf{g}$ if and only if $G(X) - X\cdot G^\sigma(X) = \mathbf{g}(0)\cdot(1-X^n)$. The verifier can query $G(X)$ and $G^\sigma(X)$ at a random point $r \in \mathbb{F}$ and check $G(r) - r\cdot G^\sigma(r) = \mathbf{g}(0)\cdot(1-r^n)$. However, in HyperPlonk, all the polynomial oracles are multilinear extensions of some vectors, we cannot use a similar method for verification. We use a different approach to perform the cyclic shift left check.

Let $\mathrm{cnext} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be a function such that $\mathrm{cnext}(\mathbf{i},\mathbf{j}) = 1$ if and only if $\mathbf{j} = \mathbf{i} + 1 \mod N$, where both $\mathbf{i}$ and $\mathbf{j}$ are represented in binary. It is easy to show that $\mathbf{g}^\sigma$ is a cyclic shift left vector of $\mathbf{g}$ if and only if for all $\mathbf{i} \in \{0,1\}^n$,

$$\mathbf{g}^\sigma(\mathbf{i}) = \sum_{\mathbf{j}\in\{0,1\}^n} \mathrm{cnext}(\mathbf{i},\mathbf{j})\mathbf{g}(\mathbf{j}). \tag{10}$$

This is equivalent to

$$\widehat{\mathbf{g}}^\sigma(\mathbf{X}) = \sum_{\mathbf{j}\in\{0,1\}^n} \widehat{\mathrm{cnext}}(\mathbf{X},\mathbf{j})\widehat{\mathbf{g}}(\mathbf{j}), \tag{11}$$

where $\widehat{\mathbf{g}}^\sigma$, $\widehat{\mathbf{g}}$, $\widehat{\mathrm{cnext}}$ are multiliear extensions of $\mathbf{g}^\sigma$, $\mathbf{g}$ and cnext. The verifier first samples a random vector $\boldsymbol{\rho} \in \mathbb{F}^n$ and queries $\widehat{\mathbf{g}}^\sigma$ at $\boldsymbol{\rho}$. Suppose $\widehat{\mathbf{g}}^\sigma(\boldsymbol{\rho}) = \gamma$, the prover and verifier then run a SumCheck protocol to ensure that $\gamma = \sum_{\mathbf{j}\in\{0,1\}^n} \widehat{\mathrm{cnext}}(\boldsymbol{\rho},\mathbf{j})\widehat{\mathbf{g}}(\mathbf{j})$. At the end of the SumCheck protocol, the verifier needs to compute the values $\widehat{\mathbf{g}}(\mathbf{r})$ and $\widehat{\mathrm{cnext}}(\boldsymbol{\rho},\mathbf{r})$ at a random point $\mathbf{r} = (r_1,\cdots,r_n) \in \mathbb{F}^n$. The value of $\widehat{\mathbf{g}}(\mathbf{r})$ can be obtained by querying the polynomial oracle $\widehat{\mathbf{g}}$, but $\widehat{\mathrm{cnext}}(\boldsymbol{\rho},\mathbf{r})$ must be calculated by the verifier. Fortunately, Setty et al. [21] provided an explicit expression similar to the $\widehat{\mathrm{cnext}}$ function, which can be computed in $O(\log N)$ time. With a slight modification to their expression, we obtain an explicit expression for $\widehat{\mathrm{cnext}}$, which can also be computed in $O(\log N)$ time. Based on this fact, the verifier can complete the verification in $O(\log N)$ time, while the time complexity for the prover in the streaming model is $O(N\log N)$ and the space complexity is $O(\log N)$. See Section 4.3 for the details of cyclic shift left check protocol.

**Proving wiring indentity constraints.** In HyperPlonk, the wiring identity constraints are captured by a permutation $\boldsymbol{\tau} : \{0,1\}^{2+n} \to \{0,1\}^{2+n}$. Let $\mathbf{w} = \mathbf{l}||\mathbf{r}||\mathbf{o}||\mathbf{0} \in \mathbb{F}^{4N}$ be the computation trace of a circuit, for checking the wiring indentity constraints, it is suffice to check that for all $\mathbf{v}' \in \{0,1\}^{2+n}$, $\mathbf{w}(\mathbf{v}') = \mathbf{w}(\boldsymbol{\tau}(\mathbf{v}'))$. We refer to this type of check as prescribed permutation check. This protocol is essentially the same as HyperPlonk (the original Plonk protocol also uses the same idea) and is suitable for implementation in the streaming model. See Section 4.5 and 4.6 for the details of prescribed permutation protocol.

**Multilinear KZG scheme in the streaming model.** We use multiliear KZG scheme [14] to construct our SNARK. Let $\mathbf{f}$ be a vector with length $N = 2^n$ and $\widehat{\mathbf{f}}$ be its multilinear extension. The prover sends a commitment $C$ about the polynomial $\widehat{\mathbf{f}}$. The verifier selects a point $\mathbf{z} \in \mathbf{F}^n$ and sends it to the prover, who then returns a value $\mu$ along with a proof $\pi$ to demonstrate that $\mu = \widehat{\mathbf{f}}(\mathbf{z})$. Note that the KZG scheme required a trusted setup to generate the *structured reference strings* (SRS). The setup algorithm first samples a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g, h, e)$, where $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| = p$, $g$ generates $\mathbb{G}_1$, $h$ generates $\mathbb{G}_2$, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a non-degenerate bilinear map, then constructs SRS as follows:

- samples a random vector $\boldsymbol{\tau} = (\tau_1, \cdots, \tau_n) \in \mathbb{F}^n$.
- constructs a vector $\mathbf{T}_{\boldsymbol{\tau}} \in \mathbb{F}^N$ $(N = 2^n)$ such that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{T}_{\boldsymbol{\tau}}(\mathbf{v}) = \widehat{\mathrm{eq}}(\boldsymbol{\tau}, \mathbf{v})$.
- outputs the commitment key $\mathrm{ck} = \mathbf{\Sigma} = \left(g^{\mathbf{T}_{\boldsymbol{\tau}}(0)}, g^{\mathbf{T}_{\boldsymbol{\tau}}(1)}, \cdots, g^{\mathbf{T}_{\boldsymbol{\tau}}(N-1)}\right)$, receiver key $\mathrm{rk} = \mathbf{\Sigma}' = (h^{\tau_1}, \cdots, h^{\tau_n})$.

The commitment to $\widehat{\mathbf{f}}$ is computed as $C = \prod_{0 \le j \le N-1} \left(g^{\mathbf{T}_{\boldsymbol{\tau}}(j)}\right)^{\mathbf{f}(j)} = g^{\widehat{\mathbf{f}}(\boldsymbol{\tau})}$. Suppose the verifer wants to know the evaluation of the polynomial at a point $\mathbf{z} \in \mathbb{F}^n$, the opening algorithm is based on the following theorem.

**Theorem 2.2.** *Suppose* $\mathbf{f} \in \mathbb{F}^N$ *is a vector of length* $N = 2^n$, $\widehat{\mathbf{f}}$ *is the multilinear extension of* $\mathbf{f}$, *then for any* $\mathbf{z} = (z_1, \cdots, z_n) \in \mathbb{F}^n$, $\widehat{\mathbf{f}}(\mathbf{z}) = \mu$ *if and only if there exists a unique set of n vectors* $\mathbf{f}_1 \in \mathbb{F}^{N/2}, \cdots, \mathbf{f}_i \in \mathbb{F}^{N/2^i}, \cdots, \mathbf{f}_n \in \mathbb{F}$ *such that*

$$\widehat{\mathbf{f}}(X_1, \cdots, X_n) - \mu = \sum_{i=1}^{n} (X_{n-i+1} - z_{n-i+1}) \cdot \widehat{\mathbf{f}}_i(X_1, \cdots, X_{n-i}), \tag{12}$$

*where* $\widehat{\mathbf{f}}_i$ *is the multilinear extension of* $\mathbf{f}_i$, *and* $\widehat{\mathbf{f}}_n$ *is a constant function with no variable.*

Define $\mathbf{f}^{(i)} \in \mathbb{F}^{N/2^i}$ such that for all $\mathbf{v} = (v_1, \cdots, v_{n-i}) \in \{0,1\}^{n-i}$,

$$\mathbf{f}^{(i)}(\mathbf{v}) = \widehat{\mathbf{f}}(v_1, \cdots, v_{n-i}, z_{n-i+1}, \cdots, z_n),$$

note that $\mathbf{f}^{(0)} = \mathbf{f}$ and $\mathbf{f}^{(n)} = \widehat{\mathbf{f}}(\mathbf{z})$. It is not hard to show that for all $\mathbf{v} \in (v_1, \cdots, v_{n-i})$,

$$\mathbf{f}_i(v_1, \cdots, v_{n-i}) = \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 1) - \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 0). \tag{13}$$

Therefore, the prover first computes $\mathbf{f}_1, \cdots, \mathbf{f}_n$ based on Theorem 2.2 and then computes $y_i = g^{\widehat{\mathbf{f}}_i(\tau)}$ for all $1 \le i \le n$ using commitment key $\Sigma$. The prover sends $\pi = (y_1, \cdots, y_n)$ and $\mu = \widehat{\mathbf{f}}(\mathbf{z})$ to the verifier. The verifer checks that

$$e(C \cdot g^{-\mu}, h) = \prod_{i=1}^{n} e(y_i, h^{\tau_{n-i+1}} \cdot h^{-z_{n-i+1}})$$

using receiver key $\Sigma'$.

Implementing the commitment and opening algorithm with low space complexity is not difficult. In the streaming model, we assume that the prover receives the streams $\mathcal{S}(\mathbf{\Sigma})$ and $\mathcal{S}(\mathbf{f})$ as inputs. Computing the commitment $C = \prod_{0 \le j \le N-1} \left(g^{\mathbf{T}_{\tau}(j)}\right)^{\mathbf{f}(j)} = g^{\widehat{\mathbf{f}}(\tau)}$ only requires making one pass over streams $\mathcal{S}(\mathbf{\Sigma})$ and $\mathcal{S}(\mathbf{f})$. Equation (13) shows that the stream $\mathcal{S}(\mathbf{f}_i)$ can be obtained from the stream $\mathcal{S}\left(\mathbf{f}^{(i-1)}\right)$ because each element in $\mathbf{f}_i$ is the difference of adjacent elements in $\mathbf{f}^{(i-1)}$. As discussed earlier in the SumCheck protocol, the stream $\mathcal{S}\left(\mathbf{f}^{(i-1)}\right)$ can be generated from the input stream $\mathcal{S}(\mathbf{f})$. Therefore, each stream $\mathcal{S}(\mathbf{f}_i)$ for $1 \le i \le n$ can be computed from the stream $\mathcal{S}(\mathbf{f})$. It follows that the proof $\pi = (y_1, \cdots, y_n)$ can be computed with low space complexity in the streaming model. See Section 6 for the details of multilinear KZG scheme.

**Comparison with Gemini.** Gemini constructs a SNARK for R1CS in the streaming model. An R1CS instance consists of three sparse matrices $A, B, C \in \mathbb{F}^{N \times N}$ and it is satisfiable if there exists a vector $\mathbf{z} \in \mathbb{F}^N$

such that $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$. Let $\mathbf{z}_A = A\mathbf{z}$, $\mathbf{z}_B = B\mathbf{z}$, $\mathbf{z}_C = C\mathbf{z}$. To verify that an R1CS instance is satisfiable, two checks need to be performed:

- **RowCheck:** check that $\mathbf{z}_A \circ \mathbf{z}_B = \mathbf{z}_C$;
- **LinCheck:** check that $\mathbf{z}_M = M\mathbf{z}$ for all $M \in \{A, B, C\}$.

Note that $\mathbf{z}_A \circ \mathbf{z}_B = \mathbf{z}_C$ if and only if $\widehat{\mathbf{z}}_A(\mathbf{v}) \cdot \widehat{\mathbf{z}}_B(\mathbf{v}) - \widehat{\mathbf{z}}_C(\mathbf{v}) = 0$ for all $\mathbf{v} \in \{0,1\}^n$, which can be checked from ZeroCheck. For checking $\mathbf{z}_M = M\mathbf{z}$ for all $M \in \{A, B, C\}$, it is suffice to check that $\langle \mathbf{y}, \mathbf{z}_M \rangle = \langle \mathbf{y}, M\mathbf{z} \rangle$ for a random vector $\mathbf{y}$. We can set $\mathbf{y} = \mathbf{T}_\rho$, where $\mathbf{T}_\rho(\mathbf{v}) = \widehat{\text{eq}}(\boldsymbol{\rho}, \mathbf{v})$ for all $\mathbf{v} \in \{0,1\}^n$. The verifier first queries $\widehat{\mathbf{z}}_M$ at point $\boldsymbol{\rho}$ and suppose the returned value is $\gamma = \widehat{\mathbf{z}}_M(\boldsymbol{\rho}) = \left\langle \mathbf{T}_\rho, \mathbf{z}_M \right\rangle$. Let $\boldsymbol{\rho}_M = M^T \mathbf{T}_\rho$, then $\langle \boldsymbol{\rho}_M, \mathbf{z} \rangle = \left\langle \mathbf{T}_\rho, M\mathbf{z} \right\rangle$. The verifier then invokes the SumCheck protocol to ensure that $\gamma = \sum_{v \in \{0,1\}^n} \boldsymbol{\rho}_M(\mathbf{v}) \cdot \mathbf{z}(\mathbf{v})$. The vector $\mathbf{T}_\rho$ can be constructed in $O(N)$ time. Since $M$ is a sparse matrix (with $O(N)$ non-zero entries), the vector $\boldsymbol{\rho}_M$ can also be constructed in $O(N)$ time. It follows that in the time-efficient model, the prover's time complexity is $O(N)$.

In the streaming model, the input streams include witness stream $\mathcal{S}(\mathbf{z})$, matrix streams $\mathcal{S}_{\text{rmaj}}(M)$, $\mathcal{S}_{\text{cmaj}}(M)$ [1] for all $M \in \{A, B, C\}$, computation trace streams $\mathcal{S}(\mathbf{z}_A)$, $\mathcal{S}(\mathbf{z}_B)$ and $\mathcal{S}(\mathbf{z}_C)$. However, in the streaming model, the prover can't store the vector $\boldsymbol{\rho}_M$ in memory. Instead, the prover can construct the stream $\mathcal{S}(\boldsymbol{\rho}_M)$ from $\mathcal{S}_{\text{rmaj}}(M)$, $\mathcal{S}_{\text{cmaj}}(M)$ and $\mathbf{T}_\rho$. Since each element of the vectors in $\mathbf{T}_\rho$ requires $O(\log N)$ time to construct and $M$ is sparse, one pass over the stream $\mathcal{S}(\boldsymbol{\rho}_M)$ requires $O(N \log N)$ time. As we discussed in the SumCheck protocol, the prover makes $O(\log N)$ passes over the stream $\mathcal{S}(\boldsymbol{\rho}_M)$. Therefore, the total run time of the LinCheck is $O(N \log^2 N)$ in the streaming model.

## 3 Preliminaries

### 3.1 Notations

We use the notation $[n]$ to denote the set $\{1, 2, \cdots, n\}$ and $\{0,1\}^n$ to denote the set of binary strings with length $n$. We use bold letters (e.g., $\mathbf{f}$) to represent a vector. For a vector $\mathbf{f}$ over $\mathbb{F}$ with length $N$, let $f_i$ denote the $i$-th entry of $\mathbf{f}$. If $N = 2^n$ for some integer $n$, we can also view $\mathbf{f}$ as a function $\mathbf{f} : \{0,1\}^n \to \mathbb{F}$. Let $0 \le v < N$, $\mathbf{v} = (v_1, \cdots, v_n) = \textbf{toBinary}(v)$ is the binary representation of $v$, where $v_1$ is the most significant bit and $v_n$ is the least significant bit, then the $v$-th element of $\mathbf{f}$ can be expressed as $\mathbf{f}(\mathbf{v}) = \mathbf{f}(v_1, \cdots, v_n) = f_v$. Similarly, if $(v_1, \cdots, v_n) \in \{0,1\}^n$, let $\textbf{toDecimal}(\mathbf{v}) = \textbf{toDecimal}(v_1, \cdots, v_n) = \sum_{i=1}^n 2^{n-i} \cdot v_i$ be its decimal represention. We will ignore the function **toBinary** and **toDecimal** when it is clear from the context, and we use the bold letter $\mathbf{v}$ to denote the binary representation (since it is a vector) and the regular letter $v$ to denote the decimal representation. Note that we have two different ways to represent the $v$-th element of $\mathbf{f}$, $f_v$ and $\mathbf{f}(\mathbf{v})$.

We use $\lambda \in \mathbb{N}$ to denote the security parameter. We use $\text{negl}(\lambda)$ to represent the negligible function with respect to $\lambda$, where $\text{negl}(\lambda) \le \lambda^{-\omega(1)}$. Given a field, we use the standard big $O(\cdot)$ notation to represent the asymptotic arithmetic complexity. where a single operation in the field constitutes one atomic operation.

[1]$\mathcal{S}_{\text{rmaj}}(M)$ ($\mathcal{S}_{\text{cmaj}}(M)$) denotes the sequence of elements in the matrix $M$ ordered in row (column) major.

Suppose $t(n)$ is a function of $n$, we use the notation $O_\lambda(t(n))$ to denote $O(t(n)\cdot\lambda^c)$ for some constant $c$, this will be useful when we describe the cryptographic operations.

## 3.2 Relations and Languages

A **relation** $\mathcal{R}$ is a set of tuples $(\mathbb{x},\mathbb{w})$ where $\mathbb{x}$ is the instance, $\mathbb{w}$ is the witness. Let $\mathcal{L}(\mathcal{R})$ be the corresponding language, i.e., $\mathcal{L}(\mathcal{R}) = \{\mathbb{x} : \exists\mathbb{w},\ \text{such that } (\mathbb{x},\mathbb{w}) \in \mathcal{R}\}$.

An **indexed relation** $\mathcal{R}$ is a set of triples $(\mathbb{i},\mathbb{x},\mathbb{w})$ where $\mathbb{i}$ is the index, $\mathbb{x}$ is the instance, $\mathbb{w}$ is the witness. For example, the indexed Boolean circuit satisfiability relation $\mathcal{R}$ consists of triples $(\mathbb{i},\mathbb{x},\mathbb{w})$, where $\mathbb{i}$ is the description of a Boolean circuit, $\mathbb{x}$ is the partial assignment of the input and $\mathbb{w}$ is the remaining assignment such that the output of the circuit is 1. Let $\mathcal{L}(\mathcal{R})$ be the corresponding language, i.e., $\mathcal{L}(\mathcal{R}) = \{(\mathbb{i},\mathbb{x}) : \exists\mathbb{w},\ \text{such that } (\mathbb{i},\mathbb{x},\mathbb{w}) \in \mathcal{R}\}$.

## 3.3 Multilinear Extension

**Definition 3.1.** Let $\mathbb{F}$ be a finite field, a multivariate polynomial $g \in \mathbb{F}[X_1,\cdots,X_n]$ with $n$ variables is multilinear if the degree of each variable in the polynomial is at most 1.

**Definition 3.2.** Let $\mathbb{F}$ be a finite field and let $\mathbf{g} : \{0,1\}^n \to \mathbb{F}$ be a function defined on the $n$-dimensional Boolean hypercube, the **multilinear extension** of $\mathbf{g}$ is a multilinear polynomial in $\mathbb{F}[X_1,\cdots,X_n]$, which we denote as $\widehat{\mathbf{g}}$, such that for all $\mathbf{v} \in \{0,1\}^n$, $\widehat{\mathbf{g}}(\mathbf{v}) = \mathbf{g}(\mathbf{v})$.

**Lemma 3.1.** *Let $\mathbb{F}$ be a finite field and let $\mathbf{g} : \{0,1\}^n \to \mathbb{F}$, then the multilinear extension of $\mathbf{g}$ is unique. Futhermore, the following multilinear polynomial $\widehat{\mathbf{g}}$ is the multilinear extension of $\mathbf{g}$:*

$$\widehat{\mathbf{g}}(\mathbf{X}) = \sum_{\mathbf{v}\in\{0,1\}^n} \mathbf{g}(\mathbf{v})\cdot\widehat{\mathrm{eq}}(\mathbf{X},\mathbf{v}), \tag{14}$$

*where $\mathbf{X} = (X_1,\cdots,X_n)$ and $\widehat{\mathrm{eq}}(\mathbf{X},\mathbf{v}) = \prod_{i\in[n]}\left((1-X_i)\cdot(1-v_i) + X_i\cdot v_i\right)$*

Let $\mathrm{eq} : \{0,1\}^n \times \{0,1\}^n \to \mathbb{F}$ be a function such that for all $\mathbf{x},\mathbf{y} \in \{0,1\}^n$, $\mathrm{eq}(\mathbf{x},\mathbf{y}) = 1$ if and only if $\mathbf{x} = \mathbf{y}$. It is easy to check that $\widehat{\mathrm{eq}}(\mathbf{X},\mathbf{Y}) = \prod_{i\in[n]}\left((1-X_i)\cdot(1-Y_i) + X_i\cdot Y_i\right)$ is the multilinear extension of eq. Let $\mathbf{r} = (r_1,\cdots,r_n)$ and $N = 2^n$, we define a vector $\mathbf{T_r} \in \mathbb{F}^N$ with length $N$ such that for all $\mathbf{v} \in \{0,1\}^n$ (the index of the vector is specified by a binary number):

$$\mathbf{T_r}(\mathbf{v}) = \widehat{\mathrm{eq}}(\mathbf{r},\mathbf{v}) = \prod_{i\in[n]}\left((1-r_i)\cdot(1-v_i) + r_i\cdot v_i\right).$$

It is also easy to check that $\widehat{\mathrm{eq}}(\mathbf{r},\mathbf{X})$ is the multilinear extension of $\mathbf{T_r}$. The evaluation of the polynomial $\widehat{\mathbf{g}}$ at a point $\mathbf{r} = (r_1,\cdots,r_n)$ can be viewed as the dot product of the two vectors $\mathbf{g}$ and $\mathbf{T_r}$, since

$$\widehat{\mathbf{g}}(\mathbf{r}) = \sum_{\mathbf{v}\in\{0,1\}^n} \mathbf{g}(\mathbf{v})\cdot\widehat{\mathrm{eq}}(\mathbf{r},\mathbf{v}) = \sum_{\mathbf{v}\in\{0,1\}^n} \mathbf{g}(\mathbf{v})\cdot\mathbf{T_r}(\mathbf{v}) = \langle\mathbf{g},\mathbf{T_r}\rangle$$

Furthermore, the vector $\mathbf{T_r}$ has very nice algebraic tensor structure.

**Definition 3.3.** Suppose $\mathbf{x} = (x_0,x_1,\cdots x_{N-1}) \in \mathbb{F}^N$ and $\mathbf{y} = (y_0,y_1,\cdots,y_{M-1}) \in \mathbb{F}^M$, then the tensor product of these two vecors $\mathbf{x} \otimes \mathbf{y} \in \mathbb{F}^{NM}$ is defined as:

$$\begin{aligned}\mathbf{x} \otimes \mathbf{y} &= (x_0 \cdot \mathbf{y}, x_1 \cdot \mathbf{y}, \cdots, x_{N-1} \cdot \mathbf{y}) \\ &= (x_0 y_0, x_0 y_1, \cdots, x_0 y_{M-1}, x_1 y_0, x_1 y_1, \cdots, x_1 y_{M-1}, \cdots, x_{N-1} y_0, x_{N-1} y_1, \cdots, x_{N-1} y_{M-1}).\end{aligned}$$

The following lemma follows immediately from the definition of the polynomial $\widehat{\mathrm{eq}}(\mathbf{X}, \mathbf{Y})$ and the definition of tensor product.

**Lemma 3.2.** *Let $\mathbf{r} = (r_1, r_2, \cdots, r_n)$ and $N = 2^n$, define a vector $\mathbf{T_r} \in \mathbb{F}^N$ such that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{T_r}(\mathbf{v}) = \widehat{\mathrm{eq}}(\mathbf{r}, \mathbf{v})$, then $\mathbf{T_r} = (1 - r_1, r_1) \otimes (1 - r_2, r_2) \otimes \cdots \otimes (1 - r_n, r_n)$. The vector $\mathbf{T_r}$ can be constructed with $O(N)$ field operations and $O(N)$ space complexity from $\mathbf{r}$.*

## 3.4 Schwartz-Zippel Lemma

**Lemma 3.3** (Schwartz-Zippel Lemma)**.** *Let $\mathbb{F}$ be a field, and $f : \mathbb{F}^n \to \mathbb{F}$ is a non-zero n-variate polynomial with total degree at most $d$. Then for any finite set $S \subseteq \mathbb{F}$,*

$$\underset{\mathbf{r} \leftarrow S^n}{\boldsymbol{Pr}} [f(\mathbf{r}) = 0] \leq \frac{d}{|S|}.$$

The following corollary follows from the Schwartz-Zippel Lemma, which will be useful for proving the soundness of our protocol.

**Corollary 3.4.** *Let $\mathbb{F}$ be a finite field and $\mathbf{a} \in \mathbb{F}^N$ is a non-zero vector with length $N = 2^n$ for some integer $n$. For any $\mathbf{r} = (r_1, \cdots, r_n) \in \mathbb{F}^n$, define $\mathbf{T_r} = (1 - r_1, r_1) \otimes (1 - r_2, r_2) \otimes \cdots \otimes (1 - r_n, r_n) \in \mathbb{F}^N$, then*

$$\underset{\mathbf{r} \leftarrow \mathbb{F}^n}{\boldsymbol{Pr}} [\langle \mathbf{a}, \mathbf{T_r} \rangle = 0] \leq \frac{\log N}{|\mathbb{F}|}. \tag{15}$$

## 3.5 Polynomial IOP

An IOP (Interactive Oracle Proof) [9] is a proof system that combines the features of PCP (Probabilistically Checkable Proof) [24, 25] and IP (Interactive Proof) [40, 41]. In the IOP model, the prover and verifier engage in an interactive protocol, and the prover provides some proofs in each round. The verifier can query these proofs at any location through oracle access. Polynomial IOP [10, 11] means that the proofs provided by the prover are polynomial oracles, which is defined below, and the verifer can query these polynomial oracles at any location.

**Definition 3.4.** Let $n, d \in \mathbb{N}$, an $(n, d)$**-polynomial oracle** is an oracle that computes a polynomial with $n$ variables and a maximal degree bound of $d$ in each variable.

If $p$ is a polynomial, we use the notation $[[p]]$ denote the corresponding polynomial oracle. In particular, if $\mathbf{f} \in \mathbb{F}^N$ is a vector over some field $\mathbb{F}$ and $N = 2^n$ for some integer $n$, if $\widehat{\mathbf{f}}$ is the multiliear extension of $\mathbf{f}$, we use the notation $[[\widehat{\mathbf{f}}]]$ to denote the multiliear polynomial oracle (i.e., the $(n, 1)$-polynomial oracle).

We adapt the definition of polynomial IOP from Marlin [10] and HyperPlonk [6].

Let $n, d \in \mathbb{N}$, a **polynomial IOP** for an indexed relation $\mathcal{R}$ with parameter $n, d$ is a protocol between indexer **I**, prover **P** and verifier **V**. The relation is an oracle relation, which means that $\mathbb{i}$ and $\mathbb{x}$ may contain some $(n, d)$-polynomial oracles. The protocol consists of two phases: an offline phase and an online phase.

In the offline phase, the indexer **I** receives the index $\mathbb{i}$ as input, and output some $(n, d)$-polynomial oracles. The implementation of indexer is deterministic and does not depend on any particular instance or witness.

In the online phase, the prover **P** aims to convince the verifier **V** that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$, where the prover's input is $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ and the verifier's input is $\mathbb{x}$ along with oracle access to the polynomial oracles provided by the indexer. The prover and verifier interact over many rounds. In each round, the verifier sends some random elements to the prover, the prover replies with several polynomial oracles or some non-oracle messages. The verifier can query the polynomial oracles at any desired locations. After the interaction, the verifier accepts or rejects. We say a prover is **admissible** if all polynomal oracles provided by the prover are $(n, d)$-polynomial oracles. The honest prover is required to be admissible under this definition.

Let $\mathcal{L}(\mathcal{R})$ be the corresponding language defined by $\mathcal{R}$. We say that the polynomial IOP has perfect completeness and soundness error $\epsilon$ if the following conditions hold:

- **Completeness.** For any $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$, the prover **P** convinces the verifier **V** to accept with probability 1.
- **Soundness.** For any $(\mathbb{i}, \mathbb{x}) \notin \mathcal{L}(\mathcal{R})$, for any **admissible** prover $\mathbf{P}^*$, the prover $\mathbf{P}^*$ convinces the verifier **V** to accept with probability at most $\epsilon$.

Futhermore, in order to construct SNARK, we require the polynomial IOP achieves the stronger property of **knowledge soundness** (against admissible prover).

- **Knowledge soundenss.** We say a polynomial IOP has knowledge soundenss error $\epsilon$ if there exists a polyomial-time extractor **E** such that for all index $\mathbb{i}$, instance $\mathbb{x}$ and admissible prover $\mathbf{P}^*$,
$$\mathbf{Pr}\left[(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R} \middle| \mathbb{w} \leftarrow \mathbf{E}^{\mathbf{P}^*}(\mathbb{i}, \mathbb{x}, 1^{d^n})\right] \geq \mathbf{Pr}\left[\text{Verifier accept}\right] - \epsilon.$$
The notation $\mathbf{E}^{\mathbf{P}^*}$ means that the extractor has black box access to the next message functions of prover **P** and **E** can rewind the prover **P**. Also note that the extractor receives $d^n$ in unary as input, since an $n$-variate polynomal with maximal individual degree $d$ requires $O(d^n)$ elements to specify.

For an indexd relation $\mathcal{R} = (\mathbb{i}, \mathbb{x}, \mathbb{w})$, it is not hard to show that if $\mathbb{w}$ consists only polynomials (which can be specified by some vectors) and $\mathbb{x}$ contains oracles to these polynomials, then a polynomial IOP with soundness error $\epsilon$ for $\mathcal{R}$ also has a knowledge soundenss error $\epsilon$. The proof of this fact can be found in HyperPlonk.

The measures of the efficiency for the polynomial IOP include:

- **Round complexity.** The number of rounds between the prover and verifier.
- **Message complexity.** The size of the non-oracle messages sent by the prover and verifier.
- **Query complexity.** The number of queries made by the verifier to the polynomial oracles.
- **Prover's time.** The runtime of the prover.
- **Prover's space.** The space complexity of the prover.

- **Verifier's time.** The runtime of the verifier.
- **Verifier's space.** The space complexity of the verifier.

Additionally, a polynomial IOP is called a public coin protocol if the randomness used by the verifier is made public, meaning that the verifier's random coins are visible to the prover. It is worth noting that a public coin interactive proof system can be transformed into a non-interactive one using the Fiat-Shamir transform in the random oracle model. In our work, all of the protocols described adhere to the public coin paradigm.

## 3.6 Streaming Model

This section provides a brief overview of the basic definitions of the streaming model. For more details on the streaming model, we recommend readers refer to Gemini [8].

Let $\mathbf{f} \in \mathbb{F}^N$ be a vector with length $N$, in the **random access model**, an algorithm can access $\mathbf{f}$ at any location $i \in \{0, \cdots, N-1\}$ to get $f_i$, i.e., the algorithm loads all elements of $\mathbf{f}$ into the memory. Conversely, in the **streaming model**, an algorithm can only scan the vector in the given order, i.e., the algorithm loads the elements of the vector into memory one by one.

**Definition 3.5.** Let $\mathbf{f} = (f_0, \cdots, f_{N-1}) \in \mathbb{F}^N$ be a vector with length $N$, we use the notation $\mathcal{S}(\mathbf{f})$ to denote the stream of vector $\mathbf{f}$. A streaming algorithm $P$ can perform the following two operations on a stream $\mathcal{S}(\mathbf{f})$:

- **init**: the stream $\mathcal{S}(\mathbf{f})$ creates or resets a counter $i = 0$.
- **next**: the stream $\mathcal{S}(\mathbf{f})$ returns $f_i$ (returns $\perp$ if $i = \perp$) and sets $i = i + 1$ ($i = \perp$ if $i = N - 1$).

We define the number of passes as the count of **init** operations that the streaming algorithm performs on its input streams. In other words, the number of passes represents the number of times the algorithm scans the stream.

A stream can be accessed by multiple streaming algorithms simultaneously. However, to ensure the space efficiency of the algorithm, we assume that the number of sessions for a stream is at most $O(\log N)$. An algorithm $P$ may has access to many streams $\mathcal{S}(\mathbf{f}_1), \cdots, \mathcal{S}(\mathbf{f}_k)$ and the output of $P$ can also be some streams.

**Definition 3.6.** Suppose $P$ and $P'$ are streaming algorithms, $P$ receives streams $\mathcal{S}(\mathbf{f}_1), \cdots, \mathcal{S}(\mathbf{f}_k)$ as input and the output of $P$ is also a stream, when $P'$ interacts with $P$, then $P'$ can do the following operation:

- $P'$ performs **init** to $P$, then the execution of $P$ is reset and $P$ also performs **init** to all $\mathcal{S}(\mathbf{f}_1), \cdots, \mathcal{S}(\mathbf{f}_k)$.
- $P'$ performs **next** to $P$, then $P$ yields next output and returns it to $P'$.
- when $P$ receives an operation (**init** or **next**) from $P'$ for accessing some streams $\mathcal{S}(\mathbf{f}_1), \cdots, \mathcal{S}(\mathbf{f}_k)$, $P$ performs the same operation to the corresponding stream and return the correct answer to $P'$.

**Lemma 3.5.** *If the streaming algorithm $P$ has time complexity $t_P$, space complexity $s_P$, input passes $k_P$, and the streaming algorithm $P'$ has time complexity $t_{P'}$, space complexity $s_{P'}$, input passes $k_{P'}$, then when $P'$ interacts with $P$, the whole streaming algorithm has complexity $k_{P'} \cdot t_P + t_{P'}$, space complexity $s_P + s_{P'}$, input passes $k_P \cdot k_{P'}$.*

## 4 A Toolbox for Multivariate Polynomials with Elastic Prover

### 4.1 SumCheck PIOP

**Definition 4.1.** The **SumCheck** relation $\mathcal{R}_{SUM}$ is the set of tuples

$$(\mathbb{x}, \mathbb{w}) = \left(\left(\mathbb{F}, N, t, d, F, [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]], \gamma\right), (\mathbf{a}_1, \cdots, \mathbf{a}_t)\right),$$

where

- $N, t, d \in \mathbb{N}$, $N = 2^n$ for some integer $n$, and $t$, $d$ are constants.
- $F(\mathbf{X}) \in \mathbb{F}[X_1, \cdots, X_t]$ is a multivariate polynomials with $t$ variables and total degree at most $d$.
- $\mathbf{a}_1, \cdots, \mathbf{a}_t \in \mathbb{F}^N$ are $t$ vectors with length $N$,
- $[[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]]$ are polynomial oracles which compute $\widehat{\mathbf{a}}_1, \cdots, \widehat{\mathbf{a}}_t$, the multilinear extensions of $\mathbf{a}_1, \cdots, \mathbf{a}_t$.
- $\sum_{\mathbf{v}\in\{0,1\}^n} F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right) = \gamma$.

**Theorem 4.1.** *There is a polynomial IOP for relation $\mathcal{R}_{SUM}$ with the following properties:*

**Table 2:** Properties of SumCheck protocol

| round complexity | message complexity | query complexity | soundness error | verifier's time |
|---|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(1)$ | $O\left(\frac{\log N}{\lvert\mathbb{F}\rvert}\right)$ | $O(\log N)$ |

*Moreover,*

- *in the **random access model**, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$,*
- *in the **streaming model**, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over all input streams.*

**Protocol 1.** SumCheck protocol:
*Inputs.*

- Prover's input:
  - instance $\mathbb{x} = \left(\mathbb{F}, N, t, d, F, [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]], \gamma\right)$,
  - witness $\mathbb{w} = (\mathbf{a}_1, \cdots, \mathbf{a}_t)$.
- Verifier's input:
  - instance $\mathbb{x} = \left(\mathbb{F}, N, t, d, F, [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]], \gamma\right)$.

*Goal.*

- The prover convinces the verifer that $\sum_{\mathbf{v}\in\{0,1\}^n} F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right) = \gamma$.

*The protocol:*

1. Let $n = \log N$, the prover constructs a polynomial

$$g(\mathbf{X}) = F\left(\widehat{\mathbf{a}}_1(\mathbf{X}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{X})\right),$$

where $\mathbf{X} = (X_1, \cdots, X_n)$.

2. In the first (1-st) round, the prover constructs and sends a univariate polynomial $g_n(X) \in \mathbb{F}[X]$, which is defined as:

$$g_n(X_n) = \sum_{v_1,\cdots,v_{n-1}\in\{0,1\}} g(v_1,\cdots,v_{n-1},X_n).$$

   The verifer first checks that $g_n(0) + g_n(1) = \gamma$ and reject if this equation does not hold. Then, the verifier samples a uniformly random element $r_n \leftarrow \mathbb{F}$ and sends it to the prover.
3. In the $l$-th round ($1 < l < n$), the prover constructs and sends a univariate polynomial $g_{n+1-l}(X_{n+1-l}) \in \mathbb{F}[X]$, which is defined as:

$$g_{n+1-l}(X_{n+1-l}) = \sum_{v_1,\cdots,v_{n-l}\in\{0,1\}} g(v_1,\cdots,v_{n-l},X_{n+1-l},r_{n+2-l},\cdots,r_n).$$

   The verifer first checks that $g_{n+1-l}(0) + g_{n+1-l}(1) = g_{n+2-l}(r_{n+2-l})$, where $g_{n+2-l}$ is sent in the $l-1$ round by the prover and $r_{n+2-l}$ is sampled in the $l-1$ round by the verifier. The verifier rejects if this equation does not hold. Then, the verifier samples a uniformly random element $r_{n+1-l} \leftarrow \mathbb{F}$ and sends it to the prover.
4. In the final ($n$-th) round, the prover constructs and sends a univariate polynomial $g_1(X) \in \mathbb{F}[X]$, which is defined as:

$$g_1(X_1) = g(X_1,r_2,\cdots,r_{n-1},r_n).$$

   The verifer first checks that $g_1(0) + g_1(1) = g_2(r_2)$, where $g_2$ is sent in the $n-1$ round by the prover and $r_2$ is sampled in the $n-1$ round by the verifier. The verifier rejects if this equation does not hold. Next, the verifier samples a uniformly random element $r_1 \leftarrow \mathbb{F}$ and computes $g_1(r_1)$. Then the verifier queries $[[\widehat{\mathbf{a}}_1]],\cdots,[[\widehat{\mathbf{a}}_t]]$ at $\mathbf{r} = (r_1,\cdots,r_n)$, suppose the returned values are $\gamma_1,\cdots,\gamma_t$, the verifier checks that $g_1(r_1) = F(\gamma_1,\cdots,\gamma_t)$.

**Lemma 4.2.** *The Protocol 1 has perfect completeness.*

**Proof.** The completness follows from the description of the protocol. □

**Lemma 4.3.** *The Protocol 1 has soundness error* $1 - \left(1 - \frac{d}{|\mathbb{F}|}\right)^n \le O\left(\frac{\log N}{|\mathbb{F}|}\right)$.

**Proof.** Suppose that $\sum_{\mathbf{v}\in\{0,1\}^n} F\left(\widehat{\mathbf{a}}_1(\mathbf{v}),\cdots,\widehat{\mathbf{a}}_t(\mathbf{v})\right) \neq \gamma$, then it holds that

$$\sum_{\mathbf{v}\in\{0,1\}^n} g(\mathbf{v}) \neq \gamma.$$

since $F$ is a multivariate polynomial with total degree at most $d$ and $\widehat{\mathbf{a}}_1,\cdots,\widehat{\mathbf{a}}_t$ are multilinear, it follows that the individual degree of each variable of $g$ is at most $d$. Suppose in each round, the malicious prover sends a polynomial $g^*_{n+1-l}(X) \in \mathbb{F}[X]$ with degree at most $d$. (If the degree of $g^*_{n+1-l}$ is greater than $d$ (more that $d+1$ coefficients), the verifier will reject immediately.) We prove the soundness error by induction on $n$.

- The base case is when $n = 1$. In this case the prover sends a polynomial $g^*_1(X)$ which is claimed to be equal to $g(X)$. If $g^*_1(X)$ is actually equal to $g(X)$, then the verifier will reject, since

$$g^*_1(0) + g^*_1(1) = g(0) + g(1) \neq \gamma.$$

If $g_1^*(X)$ is not equal to $g(X)$, then from Schwartz-Zipple Lemma, for a uniformly random element $r$, the test $g_1^*(r) = g(r)$ will pass with probability at most $\frac{d}{|\mathbb{F}|} = 1 - \left(1 - \frac{d}{|\mathbb{F}|}\right)$.

- The inductive step is when $n > 1$, we assume that the soundness error holds for $n - 1$. Suppose the verifer accepts in the first round (otherwise we are done), then the polynomial $g_n^*(X)$ sent by the prover is not equal to $g_n(X)$, otherwise

$$g_n^*(0) + g_n^*(1) = g_n(0) + g_n(1) = \sum_{\mathbf{v} \in \{0,1\}^n} g(\mathbf{v}) \neq \gamma,$$

the verifier will reject in the first round. Let $E_1$ be the event that $g_n^*(r_n) \neq g_n(r_n)$, then from the Schwartz-Zipple Lemma, $\mathbf{Pr}[E_1] \geq 1 - \frac{d}{|\mathbb{F}|}$. Let $E_2$ be the event that the verifier rejects in $i \geq 2$ round. By induction hypothesis, we know that $\mathbf{Pr}[E_2|E_1] \geq \left(1 - \frac{d}{|\mathbb{F}|}\right)^{n-1}$, since $g_n^*(r_n) \neq g_n(r_n) = \sum_{v_1,\cdots,v_{n-1} \in \{0,1\}} g(v_1, v_2, \cdots, v_{n-1}, r_n)$. Therefore,

$$\begin{aligned}\mathbf{Pr}[\text{Verifier accepts}] &= 1 - \mathbf{Pr}[\text{Verifier rejects}] \\ &= 1 - \mathbf{Pr}[E_2] \\ &\leq 1 - \mathbf{Pr}[E_2|E_1] \cdot \mathbf{Pr}[E_1] \\ &\leq 1 - \left(1 - \frac{d}{|\mathbb{F}|}\right)^n .\end{aligned}$$

The second equation holds since we assume that the verifier always accepts in the first round.

□

**Lemma 4.4.** *In the Protocol 1, the round complexity is $O(\log N)$, message complexity is $O(\log N)$, query complexity is $O(1)$ and the verifier has arithmetic complexity $O(\log N)$.*

**Proof.** The round complexity is immediately from the protocol and $n = \log N$. In each round, the prover sends an univariate polynomial with degree at most $d$, which can be specified by at most $d + 1$ elements. The verifier sends a random element in each round, Therefore, the total message complexity is $(d + 1) \cdot \log N = O(\log N)$ since we assume $d$ is a constant. At the end of the protocol, the verifier should query the $t$ polynomial oracles $[[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]]$ at a random point $\mathbf{r} = (r_1, \cdots, r_n)$, therefore the query complexity is $O(1)$ since we assume $t$ is a constant.

In each round, the verifier receives a polynomial $g_{n+1-l}(X_{n+1-l})$, the verifier will sample a random element and computes $g_{n+1-l}(r_{n+1-l})$, this can be done using $O(1)$ field operations. In the last step, the verifier checks $g_1(r_1) = F(\gamma_1, \cdots, \gamma_t)$ using $O(1)$ field operations. Totally, the arithmetic complexity for the verifier is $O(\log N)$. □

We show how to implement the prover in both random access model and streaming model. In each round, the prover sends a univariate polynomial $g_{n+1-l}(X_{n+1-l})$ with degree at most $d$. The polynomial $g_{n+1-l}(X_{n+1-l})$ is defined as follows:

$$g_{n+1-l}(X_{n+1-l}) = \sum_{\mathbf{v} \in \{0,1\}^{n-l}} F\left(\widehat{\mathbf{a}}_1(\mathbf{v}, X_{n+1-l}, \mathbf{r}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v}, X_{n+1-l}, \mathbf{r})\right). \quad (16)$$

where $\mathbf{v} = (v_1, \cdots, v_{n-l})$ and $\mathbf{r} = (r_{n+2-l}, \cdots, r_n)$.

**Definition 4.2.** Let $\mathbf{a} \in \mathbb{F}^N$ and $N = 2^n$ for some integer $n$, $\mathbf{r} = (r_1, \cdots, r_n) \in \mathbb{F}^n$, suppose $\widehat{\mathbf{a}}$ is the multilinear extension of $\mathbf{a}$, for all $0 \le l \le n$, define a vector $\mathbf{a}^{(l)} \in \mathbb{F}^{N/2^l}$ such that for all $\mathbf{v} \in \{0,1\}^{n-l}$,

$$\mathbf{a}^{(l)}(\mathbf{v}) = \widehat{\mathbf{a}}(v_1, \cdots, v_{n-l}, r_{n-l+1}, \cdots, r_n),$$

note that $\mathbf{a}^{(0)} = \mathbf{a}$ and $\mathbf{a}^{(n)} = \widehat{\mathbf{a}}(r_1, \cdots, r_n)$.

**Lemma 4.5.** *Let $\mathbf{a}^{(l)} \in \mathbb{F}^{N/2^l}$ and $\mathbf{a}^{(l-1)} \in \mathbb{F}^{N/2^{(l-1)}}$ be two vectors defined from Definition 4.2, then for all $\mathbf{v} \in \{0,1\}^{n-l}$:*

$$\mathbf{a}^{(l)}(\mathbf{v}) = (1 - r_{n+1-l}) \cdot \mathbf{a}^{(l-1)}(v_1, \cdots, v_{n-l}, 0) + r_{n+1-l} \cdot \mathbf{a}^{(l-1)}(v_1, \cdots, v_{n-l}, 1)$$

**Proof.**

$$\begin{aligned}\mathbf{a}^{(l)}(\mathbf{v}) &= \widehat{\mathbf{a}}(v_1, \cdots, v_{n-l}, r_{n-l+1}, \cdots, r_n)\\ &= (1 - r_{n+1-l}) \cdot \widehat{\mathbf{a}}(v_1, \cdots, v_{n-l}, 0, r_{n-l+2}, \cdots, r_n) + r_{n+1-l} \cdot \widehat{\mathbf{a}}(v_1, \cdots, v_{n-l}, 1, r_{n-l+2}, \cdots, r_n)\\ &= (1 - r_{n+1-l}) \cdot \mathbf{a}^{(l-1)}(v_1, \cdots, v_{n-l}, 0) + r_{n+1-l} \cdot \mathbf{a}^{(l-1)}(v_1, \cdots, v_{n-l}, 1).\end{aligned}$$

The second equality holds because $\widehat{\mathbf{a}}$ is multilinear. □

**Lemma 4.6.** *In the Protocol 1, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$ in the random access model.*

**Proof.** The prover uses the Algorithm 1 to compute $g_{n+1-l}(X)$ in each round. In the $l$-th round, the prover first constructs $t$ vectors $\mathbf{a}_j^{(l-1)} \in \mathbb{F}^{N/2^{l-1}}$ from $\mathbf{a}_j^{(l-2)}$ for all $j \in [t]$ (lines 9 - 10). Remark that $\mathbf{a}_j^{(0)} = \mathbf{a}_j$, so there is no need to construct $\mathbf{a}_j^{(0)}$ in the first round. In line 12, the prover computes a linear polynomial $h_j(X)$, which is

$$\begin{aligned}h_j(X) &= (1 - X) \cdot \mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 0) + X \cdot \mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 1)\\ &= (1 - X) \cdot \widehat{\mathbf{a}}_j(v_1, \cdots, v_{n-l}, 0, r_{n-l+2}, \cdots, r_n) + X \cdot \widehat{\mathbf{a}}_j(v_1, \cdots, v_{n-l}, 1, r_{n-l+2}, \cdots, r_n)\\ &= \widehat{\mathbf{a}}_j(v_1, \cdots, v_{n-l}, X, r_{n-l+2}, \cdots, r_n).\end{aligned}$$

From line 14, it can be observed that after iterating $N/2^l$ times, the final result of $g_{n+1-l}(X)$ is

$$\sum_{v_1,\ldots,v_{n-l} \in \{0,1\}} F\left(\widehat{\mathbf{a}}_1(v_1, \cdots, v_{n-l}, X, r_{n-l+2}, \cdots, r_n), \cdots, \widehat{\mathbf{a}}_t(v_1, \cdots, v_{n-l}, X, r_{n-l+2}, \cdots, r_n)\right),$$

which is the same as Equation 16. Therefore, the output of the prover in each round is correct from Algorithm 1.

In the $l$-th round, the arithmetic complexity of prover is $O(2^{n-l})$, since we assume $t$ and $d$ are constant, the arithmetic complexity of computing $F\left(h_1(X), \cdots, h_t(X)\right)$ is $O(1)$. Therefore, the total arithmetic complexity of prover is $\sum_{l=1}^{n} O(2^{n-l}) = O(2^n) = O(N)$.

The space complexity of the prover is $O(N)$, as in each round, the prover needs to compute $t$ vectors with a length of $2^{n-l+1}$ and store them in memory. □

**Lemma 4.7.** *In the Protocol 1, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over all input streams $\mathcal{S}(\boldsymbol{a}_1), \cdots, \mathcal{S}(\boldsymbol{a}_t)$ in the streaming model.*

**Algorithm 1:** Prover's algorithm of Protocol 1 in random access model

```
Input: An instance-witness pair $(\mathbb{x}, \mathbb{w})$, all the data is stored in memory.
    • instance $\mathbb{x} = \big(\mathbb{F}, N, t, d, F, [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]], \gamma\big)$,
    • witness $\mathbb{w} = (\mathbf{a}_1, \cdots, \mathbf{a}_t)$.
Output: $g_{n+1-l}(X)$ in each $(1 \le l \le n)$ round.
1  for $l \leftarrow 1$ to $n$ do
2      if $l \neq 1$ then
3          Init $t$ vectors $\mathbf{a}_j^{(l-1)}$ with length $N/2^{l-1} = 2^{n-l+1}$ for each $j \in [t]$;
4      end
5      $g_{n+1-l}(X) \leftarrow 0$ ;                // Init the polynomial with degree $d$.
6      for $v \leftarrow 0$ to $N/2^l - 1$ do
           // Suppose $(v_1, \cdots, v_{n-l})$ is the binary representation of $v$.
7          for $j \leftarrow 1$ to $t$ do
8              if $l \neq 1$ then
9                  $\mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 0) \leftarrow$
                     $(1 - r_{n+2-l}) \cdot \mathbf{a}_j^{(l-2)}(v_1, \cdots, v_{n-l}, 0, 0) + r_{n+2-l} \cdot \mathbf{a}_j^{(l-2)}(v_1, \cdots, v_{n-l}, 0, 1)$;
10                 $\mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 1) \leftarrow$
                     $(1 - r_{n+2-l}) \cdot \mathbf{a}_j^{(l-2)}(v_1, \cdots, v_{n-l}, 1, 0) + r_{n+2-l} \cdot \mathbf{a}_j^{(l-2)}(v_1, \cdots, v_{n-l}, 1, 1)$;
                   // $r_{n+2-l}$ is sent by the verifier in $(l-1)$-th round, $\mathbf{a}_j^{(l-2)}$ is constructed by
                      the prover in $(l-1)$-th round and $\mathbf{a}_j^{(0)} = \mathbf{a}_j$.
11             end
12             $h_j(X) \leftarrow (1 - X) \cdot \mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 0) + X \cdot \mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 1)$;
               // $h_j(X)$ is a linear polynomial, which can be specified with two field elements
13         end
14         $g_{n+1-l}(X) \leftarrow g_{n+1-l}(X) + F\,(h_1(X), \cdots, h_t(X))$;
15     end
16     Send $g_{n+1-l}(X)$ to the verifier;
17     Receive a random element $r_{n+1-l}$ from the verifier;
18 end
```

**Algorithm 2:** Streaming algorithm for constructing $\mathcal{S}(\mathbf{a}_j^{(l-1)})$

```
Input: Stream $\mathcal{S}(\mathbf{a}_j)$, field elements $r_n, \cdots, r_{n+2-l}$.
Output: Stream $\mathcal{S}(\mathbf{a}_j^{(l-1)})$.
init:
    Init a stack $s$ ;   // the elements of the stack $s$ are pairs $(elem, k)$, where $elem \in \mathbb{F}$ and $0 \le k \le l$, $elem$ is in the vector $a^{(k)}$.
    $\mathcal{S}(\mathbf{a}_j).init()$ ;   // init the input stream.
    $counter \leftarrow 0$;
end
next:
    while $counter < 2^{n-l+1}$ do
        if $s.size() < 2$ then
            $elem \leftarrow \mathcal{S}(\mathbf{a}_j).next()$;
            $s.push(elem, 0)$ ;   // If the stack has at most 1 element, push a new element from $\mathcal{S}(\mathbf{a}_j)$ into stack.
        else
            $(elem_1, k_1) \leftarrow s.pop()$;
            $(elem_2, k_2) \leftarrow s.pop()$ ;   // Pop two elements from the stack.
            if $k_1 = k_2$ then
                $elem \leftarrow (1 - r_{n-k_1}) \cdot elem_2 + r_{n-k_1} \cdot elem_1$;
                // If two elements from the same vector $\mathbf{a}_j^{(k_1)}$, merge them and get a element in $\mathbf{a}_j^{(k_1+1)}$
                if $k_1 == l - 2$ then
                    yield $elem$ ;   // If the merged element is in $\mathbf{a}_j^{(l-1)}$, yield it.
                    $counter++$;
                end
                $s.push(elem, k_1 + 1)$ ;   // Push the new element into stack.
            else
                $s.push(elem_2, k_2)$;
                $s.push(elem_1, k_1)$;
                $elem \leftarrow \mathcal{S}(\mathbf{a}_j).next()$;
                $s.push(elem, 0)$ ;   // If $elem_1$ and $elem_2$ come from different vectors, push them back to the stack and push a new element from $\mathcal{S}(\mathbf{a}_j)$ into stack.
            end
        end
    end
end
```

**Algorithm 3:** Prover's algorithm of Protocol 1 in streaming model

**Input:** An instance-witness pair $(\mathbb{x}, \mathbb{w})$:

- instance $\mathbb{x} = \big(\mathbb{F}, N, t, d, F, [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]], \gamma\big)$, these data is stored in memory,
- witness $\mathbb{w} = (\mathbf{a}_1, \cdots, \mathbf{a}_t)$, these vectors are all input in the form of streams $\mathcal{S}(\mathbf{a}_1), \cdots, \mathcal{S}(\mathbf{a}_t)$.

**Output:** $g_{n+1-l}(X)$ in each $(1 \le l \le n)$ round.

```
1  for $l \leftarrow 1$ to $n$ do
2      if $l \neq 1$ then
3          for $j \leftarrow 1$ to $t$ do
4              $\mathcal{S}(\mathbf{a}_j^{(l-1)}).init()$ ;          // Init $t$ streams $\mathcal{S}(\mathbf{a}_j^{(l-1)})$
5          end
6      end
7      $g_{n+1-l}(X) \leftarrow 0$ ;          // Init the polynomial with degree $d$.
8      for $v \leftarrow 0$ to $N/2^l - 1$ do
           // Suppose $(v_1, \cdots, v_{n-l})$ is the binary representation of $v$.
9          for $j \leftarrow 1$ to $t$ do
10             if $l \neq 1$ then
11                 $\mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 0) \leftarrow \mathcal{S}(\mathbf{a}_j^{(l-1)}).next()$ ;
12                 $\mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 1) \leftarrow \mathcal{S}(\mathbf{a}_j^{(l-1)}).next()$ ;
                   // $r_{n+2-l}$ is sent by the verifier in $(l-1)$-th round, $\mathcal{S}(\mathbf{a}_j^{(l-1)})$ is constructed
                      using Algorithm 2.
13             end
14             $h_j(X) \leftarrow (1-X) \cdot \mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 0) + X \cdot \mathbf{a}_j^{(l-1)}(v_1, \cdots, v_{n-l}, 1)$;
               // $h_j(X)$ is a linear polynomial, which can be specified with two field elements
15         end
16         $g_{n+1-l}(X) \leftarrow g_{n+1-l}(X) + F\,(h_1(X), \cdots, h_t(X))$;
17     end
18     Send $g_{n+1-l}(X)$ to the verifier;
19     $r_{n+1-l} \leftarrow V$ ;          // Receive a random element from verifier.
20 end
```

**Proof.** In the streaming model, the vector $\mathbf{a}_1, \cdots, \mathbf{a}_t$ are provided as $t$ streams $\mathcal{S}(\mathbf{a}_1), \cdots, \mathcal{S}(\mathbf{a}_t)$ to the prover. In Algorithm 1, the prover constructs a new vector $\mathbf{a}_j^{(l-1)}$ in the $l$-th round. In the streaming model, we can view the process of constructing this vector as building a stream $\mathcal{S}(\mathbf{a}_j^{(l-1)})$. The prover can use this stream to compute the polynomial $h_j(X)$, as the polynomial $h_j(X)$ is constructed from adjacent elements in this stream. In the $l$-th round, the prover should constructs a new vector $\mathbf{a}_j^{(l-1)}$ from $\mathbf{a}_j^{(l-2)}$. However, the vector $\mathbf{a}_j^{(l-2)}$ is no longer stored in memory, therefore, the prover should constructs the stream $\mathcal{S}(\mathbf{a}_j^{(l-1)})$ from the input stream $\mathcal{S}(\mathbf{a}_j)$. We present the streaming algorithm for constructing stream $\mathbf{a}_j^{(l-1)}$ in Algorithm 2 (remenber that a stream supports two operations: **init** and **next**). We use the keyword **yield** (from **Python**) to output the next element of a stream.

We initialize a stack $s$ to store the "rightmost" two elements of each vector. If $s$ has at most one element, then we read a element from $\mathcal{S}(\mathbf{a}_j)$ and push it to $s$. If $s$ has more than one element, we pop two elements from $s$. If this two elements $elem_1$ and $elem_2$ are in the same vector, say $\mathbf{a}_j^{(k_1)}$, then they must be adjacent. Suppose

- $elem_1 = \mathbf{a}_j^{(k_1)}(v_1, \cdots, v_{n-k_1-1}, 1) = \widehat{\mathbf{a}}_j(v_1, \cdots, v_{n-k_1-1}, 1, r_{n-k_1+1}, \cdots, r_n)$,
- $elem_2 = \mathbf{a}_j^{(k_1)}(v_1, \cdots, v_{n-k_1-1}, 0) = \widehat{\mathbf{a}}_j(v_1, \cdots, v_{n-k_1-1}, 0, r_{n-k_1+1}, \cdots, r_n)$,

for some $v_1, \cdots, v_{n-k_1} \in \mathbb{F}$ (since $elem_1$ is popped before $elem_2$, $elem_1$ is in the "right" side of $elem_2$). Then in line 15, the prover computes a new element:

$$\begin{aligned} elem &= (1 - r_{n-k_1}) \cdot elem_2 + r_{n-k_1} \cdot elem_1 \\ &= (1 - r_{n-k_1}) \cdot \widehat{\mathbf{a}}_j(v_1, \cdots, v_{n-k_1-1}, 0, r_{n-k_1+1}, \cdots, r_n) + r_{n-k_1} \cdot \widehat{\mathbf{a}}_j(v_1, \cdots, v_{n-k_1-1}, 1, r_{n-k_1+1}, \cdots, r_n) \\ &= \widehat{\mathbf{a}}_j(v_1, \cdots, v_{n-k_1-1}, r_{n-k_1}, r_{n-k_1+1}, \cdots, r_n) \\ &= \mathbf{a}_j^{(k_1+1)}(v_1, \cdots, v_{n-k_1-1}). \end{aligned}$$

If $k_1 + 1 = l - 1$, then yield $elem$ since it is in the target vector $\mathbf{a}_j^{(l-1)}$. If $k_1 + 1 \neq l - 1$, push it into the stack as it is the "rightmost" element of vector $\mathbf{a}_j^{(k_1+1)}$ at this moment. If $elem_1$ and $elem_2$ are in different vectors, then we push them back to the stack and push a new element from $\mathcal{S}(\mathbf{a}_j)$ into stack.

The space cost of Algorithm 2 is primarily from the stack. Since each vector has at most two elements in the stack (in fact, most vectors have only one element in the stack), the space complexity of the algorithm is $O(l)$. Each element in the vector $\mathbf{a}_j^{(l-1)}$ requires $O(1 + 2 + 4 + \cdots + 2^{l-2}) = O(2^{l-1})$ field operations for construction. As a result, the overall arithmetic complexity is $O(2^{l-1}) \cdot O(N/2^{l-1}) = O(N)$. Furthermore, the Algorithm 2 just makes one pass to the input stream $\mathcal{S}(\mathbf{a}_j)$.

In the streaming model, the prover invokes Algorithm 2 to generate the polynomial $g_{n+1-l}(X)$ in Protocol 1. The prover uses the Algorithm 3 to compute $g_{n+1-l}(X)$ in each round. Except for the $t$ vectors $\mathbf{a}_1, \cdots, \mathbf{a}_t$, all other data is stored in memory, as the space occupied by these data is relatively small. The processes of Algorithm 3 and Algorithm 1 are essentially the same, except that each element in vector a is derived from the stream generated by Algorithm 2. We use a yellow background to distinguish it from Algorithm 1. Therefore, the correctness of Algorithm 3 comes from the correctness of Algorithms 1 and 2.

Based on the previous analysis, the space complexity of the algorithm generating stream $\mathcal{S}(\mathbf{a}_j^{(l-1)})$ in the $l$-th round is $O(l)$. Therefore, the overall space complexity is $O(\log N)$. We have showed that

$\mathcal{S}(\mathbf{a}_j^{(l-1)}).next()$ takes $O(2^{l-1})$ field operations to yield an element, it follows that arithmetic complexity in each round is $O(2^{n-l} \cdot 2 \cdot 2^{l-1}) = O(2^n) = O(N)$. Therefore, the total arithmetic complexity for the prover is $\sum_{l=1}^{n} O(N) = O(N \cdot \log N)$. In each round, Algorithm 2 makes one pass over each streams. Therefore, in total, the prover makes $O(\log N)$ passes over each streams $\mathcal{S}(\mathbf{a}_1), \cdots, \mathcal{S}(\mathbf{a}_t)$. □

## 4.2 ZeroCheck PIOP

**Definition 4.3.** The **ZeroCheck** relation $\mathcal{R}_{ZERO}$ is the set of tuples

$$(\mathbb{x}, \mathbb{w}) = \left(\left(\mathbb{F}, N, t, d, F, [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]]\right), (\mathbf{a}_1, \cdots, \mathbf{a}_t)\right),$$

where

- $N, t, d \in \mathbb{N}$, $N = 2^n$ for some integer $n$ and $t$, $d$ are constants.
- $F(\mathbf{X}) \in \mathbb{F}[X_1, \cdots, X_t]$ is a multivariate polynomials with $t$ variables and total degree at most $d$.
- $\mathbf{a}_1, \cdots, \mathbf{a}_t \in \mathbb{F}^N$ are $t$ vectors with length $N$,
- $[[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]]$ are polynomial oracles which compute $\widehat{\mathbf{a}}_1, \cdots, \widehat{\mathbf{a}}_t$, the multilinear extensions of $\mathbf{a}_1, \cdots, \mathbf{a}_t$.
- $F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right) = 0$ for all $\mathbf{v} \in \{0, 1\}^n$.

**Theorem 4.8.** *There is a polynomial IOP for relation $\mathcal{R}_{ZERO}$ with the following properties:*

**Table 3:** Properties of ZeroCheck protocol

| round complexity | message complexity | query complexity | soundness error | verifier's time |
|---|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(1)$ | $O\left(\frac{\log N}{\lvert\mathbb{F}\rvert}\right)$ | $O(\log N)$ |

*Moreover,*

- *in the* ***random access model****, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$,*
- *in the* ***streaming model****, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over all input streams.*

**Protocol 2.** ZeroCheck protocol:
*Inputs.*

- Prover's input:
  - instance $\mathbb{x} = \left(\mathbb{F}, N, t, d, F, [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]]\right)$,
  - witness $\mathbb{w} = (\mathbf{a}_1, \cdots, \mathbf{a}_t)$.
- Verifier's input:
  - instance $\mathbb{x} = \left(\mathbb{F}, N, t, d, F, [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]]\right)$.

*Goal.*

- The prover convinces the verifer that $F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right) = 0$ for all $\mathbf{v} \in \{0, 1\}^n$.

*The protocol:*

1. Let $n = \log N$, the verifier randomly samples $\boldsymbol{\rho} = (\rho_1, \rho_2, \cdots, \rho_n) \in \mathbb{F}^n$, sends to the prover.
2. The prover constructs a vector $\mathbf{T}_{\rho} \in \mathbb{F}^N$:
$$\mathbf{T}_{\rho} = (1-\rho_1, \rho_1) \otimes (1-\rho_2, \rho_2) \otimes \cdots \otimes (1-\rho_n, \rho_n).$$
3. Let
   - $\mathbb{x}' = \big(\mathbb{F}, N, t+1, d+1, F', [[\widehat{\mathbf{a}}_1]], \cdots, [[\widehat{\mathbf{a}}_t]], [[\widehat{\mathbf{a}}_{t+1}]], 0\big)$,
   - $\mathbb{w}' = (\mathbf{a}_1, \cdots, \mathbf{a}_t, \mathbf{a}_{t+1})$,

   where $F'(X_1, \cdots, X_t, X_{t+1}) = F(X_1, \cdots, X_t) \cdot X_{t+1}$ and $\mathbf{a}_{t+1} = \mathbf{T}_{\rho}$. The prover and verifier invoke SumCheck protocol for checking that $(\mathbb{x}', \mathbb{w}') \in \mathcal{R}_{SUM}$.

Note that the verifer has virtual oracle access to $[[\widehat{\mathbf{a}}_{t+1}]] = [[\widehat{\mathbf{T}}_{\rho}]]$, since $\widehat{\mathbf{T}}_{\rho}(\mathbf{X}) = \widehat{\mathrm{eq}}(\boldsymbol{\rho}, \mathbf{X})$.

**Lemma 4.9.** *The Protocol 2 has perfect completeness.*

**Proof.** Suppose that $F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right) = 0$ for all $\mathbf{v} \in \{0,1\}^n$, then
$$\begin{aligned}\sum_{\mathbf{v}\in\{0,1\}^n} F'\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v}), \widehat{\mathbf{a}}_{t+1}(\mathbf{v})\right) &= \sum_{\mathbf{v}\in\{0,1\}^n} F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right) \cdot \widehat{\mathbf{a}}_{t+1}(\mathbf{v}) \\ &= \sum_{\mathbf{v}\in\{0,1\}^n} F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right) \cdot \widehat{\mathrm{eq}}(\boldsymbol{\rho}, \mathbf{v}) \\ &= 0.\end{aligned}$$
It follows that $(\mathbb{x}', \mathbb{w}') \in \mathcal{R}_{SUM}$. The completness of this protocol then follows from the completeness of SumCheck protocol. □

**Lemma 4.10.** *The Protocol 2 has soundness error $O\left(\frac{\log N}{|\mathbb{F}|}\right)$.*

**Proof.** Let $\mathbf{a} \in \mathbb{F}^N$ such that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{a}(\mathbf{v}) = F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right)$, suppose that $\mathbf{a} \neq \mathbf{0}$, then from Corollary 3.4, it holds that
$$\mathbf{Pr}_{\rho \leftarrow \mathbb{F}^n}\left[\left\langle \mathbf{a}, \mathbf{T}_{\rho}\right\rangle = 0\right] \leq \frac{n}{|\mathbb{F}|} = \frac{\log N}{\mathbb{F}}.$$
Suppose that $\left\langle \mathbf{a}, \mathbf{T}_{\rho}\right\rangle \neq 0$, then
$$\begin{aligned}\sum_{\mathbf{v}\in\{0,1\}^n} F'\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v}), \widehat{\mathbf{a}}_{t+1}(\mathbf{v})\right) &= \sum_{\mathbf{v}\in\{0,1\}^n} F\left(\widehat{\mathbf{a}}_1(\mathbf{v}), \cdots, \widehat{\mathbf{a}}_t(\mathbf{v})\right) \cdot \widehat{\mathbf{a}}_{t+1}(\mathbf{v}) \\ &= \sum_{\mathbf{v}\in\{0,1\}^n} \mathbf{a}(\mathbf{v}) \cdot \widehat{\mathrm{eq}}(\boldsymbol{\rho}, \mathbf{v}) \\ &= \left\langle \mathbf{a}, \mathbf{T}_{\rho}\right\rangle \neq 0.\end{aligned}$$
It follows that $(\mathbb{x}', \mathbb{w}') \notin \mathcal{R}_{SUM}$. From the soundness of SumCheck protocol, the verifier will accept with probability at most $O\left(\frac{\log N}{|\mathbb{F}|}\right)$. Therefore,
$$\begin{aligned}\mathbf{Pr}\left[\text{Verifier accepts}\right] &\leq \mathbf{Pr}\left[\text{Verifier accepts}\middle| \left\langle \mathbf{a}, \mathbf{T}_{\rho}\right\rangle \neq 0\right] + \mathbf{Pr}\left[\left\langle \mathbf{a}, \mathbf{T}_{\rho}\right\rangle = 0\right] \\ &\leq O\left(\frac{\log N}{|\mathbb{F}|}\right).\end{aligned}$$
□

**Lemma 4.11.** *In the Protocol 2, the round complexity is $O(\log N)$, message complexity is $O(\log N)$, query complexity is $O(1)$ and the verifier has arithmetic complexity $O(\log N)$.*

**Proof.** In the step 1 of this protocol, the verifier sends $\log N$ random elements to the prover. The overhead in the latter part of the protocol is all due to the SumCheck protocol. This lemma follows from Theorem 4.1. Note that $\widehat{\mathbf{a}}_{t+1}(\mathbf{X}) = \widehat{\mathrm{eq}}(\boldsymbol{\rho}, \mathbf{X})$, the verifier can computes $\widehat{\mathbf{a}}_{t+1}(\mathbf{X})$ at any points by himself using $O(\log N)$ field operations. □

**Lemma 4.12.** *In the Protocol 2, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$ in the random access model.*

**Proof.** The vector $\mathbf{T}_\rho$ can be constructed using $O(N)$ field operations and $O(N)$ space, the lemma then follows from Theorem 4.1. □

**Lemma 4.13.** *In the Protocol 2, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over all input streams $\mathcal{S}(\mathbf{a}_1), \cdots, \mathcal{S}(\mathbf{a}_t)$ in the streaming model.*

**Proof.** In the streaming model, the prover can't construct the vector $\mathbf{a}_{t+1} = \mathbf{T}_\rho$ explicitly, since it takes large space to store. Recall that in $l$-th round of the SumCheck protocol, the prover should compute $\mathbf{a}_{t+1}^{(l-1)}(\mathbf{v})$ for all $\mathbf{v} \in \{0,1\}^{n-l+1}$, each element takes $O(\log N)$ field operations to construct since

$$\mathbf{a}_{t+1}^{(l-1)}(\mathbf{v}) = \widehat{\mathrm{eq}}(\boldsymbol{\rho}, v_1, \cdots, v_{n-l+1}, r_{n-l+2}, \cdots, r_n). \tag{17}$$

We have showed that $\mathbf{a}_j^{(l-1)}(\mathbf{v})$ takes $O(2^{l-1})$ field operations to construct for $j \in [t]$ from the input stream $\mathcal{S}(\mathbf{a}_j)$. It follows that in each round of the SumCheck protocol, the arithmetic complexity is $2^{n-l} \cdot O(\log N + 2^{l-1}) = O(N \cdot \log N/2^l + N)$. Therefore, the total arithmetic complexity is $\sum_{l=1}^{n} O(N \cdot \log N/2^l + N) = O(N \log N)$. The space complexity remains $O(\log N)$. □

## 4.3 Cyclic Shift Left Check PIOP

**Definition 4.4.** The **cyclic shift left** relation $\mathcal{R}_{CSL}$ is the set of tuples

$$(\mathbb{x}, \mathbb{w}) = \left((\mathbb{F}, N, [[\widehat{\mathbf{z}}]], [[\widehat{\mathbf{z}}^\sigma]]), (\mathbf{z}, \mathbf{z}^\sigma)\right),$$

where

- $N = 2^n$ for some integer $n$,
- $\mathbf{z}, \mathbf{z}^\sigma \in \mathbb{F}^N$.
- $[[\widehat{\mathbf{z}}]], [[\widehat{\mathbf{z}}^\sigma]]$ are polynomial oracles which compute $\widehat{\mathbf{z}}, \widehat{\mathbf{z}}^\sigma$ the multilinear extensions of $\mathbf{z}, \mathbf{z}^\sigma$.
- $\mathbf{z}^\sigma$ is the cyclic shift left of $\mathbf{z}$, i.e.,

$$\mathbf{z}^\sigma = (z_1, z_2, \cdots, z_{N-1}, z_0).$$

**Theorem 4.14.** *There is a polynomial IOP for relation $\mathcal{R}_{CSL}$ with the following properties:*

*Moreover,*

- *in the **random access model**, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$,*

**Table 4:** Properties of cyclic shift left protocol

| round complexity | message complexity | query complexity | soundness error | verifier's time |
|---|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(1)$ | $O\left(\frac{\log N}{\lvert\mathbb{F}\rvert}\right)$ | $O(\log N)$ |

- *in the **streaming model**, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over all input streams.*

**Definition 4.5.** Let $N$ be a positive integer and $N = 2^n$ for some integer $n$, define cnext : $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be a function such that for all $\mathbf{i}, \mathbf{j} \in \{0,1\}^n$, $\text{cnext}(\mathbf{i}, \mathbf{j}) = 1$ if and only if $j = (i+1) \mod N$, where $i = \textbf{toDecimal}(\mathbf{i})$ and $j = \textbf{toDecimal}(\mathbf{j})$.

**Lemma 4.15.** *Let $\mathbf{z}, \mathbf{z}^\sigma \in \mathbb{F}^N$, $N = 2^n$ for some integer $n$, $\mathbf{z}^\sigma$ is the cyclic shift left of $\mathbf{z}$ if and only if for any $\mathbf{i} \in \{0,1\}^n$,*

$$\mathbf{z}^\sigma(\mathbf{i}) = \sum_{\mathbf{j} \in \{0,1\}^n} \text{cnext}(\mathbf{i}, \mathbf{j}) \cdot \mathbf{z}(\mathbf{j}). \tag{18}$$

*Moreover, Let $\widehat{\mathbf{z}}, \widehat{\mathbf{z}}^\sigma : \mathbb{F}^n \to \mathbb{F}$ and $\widehat{\text{cnext}} : \mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}$ be the multilinear extension of $\mathbf{z}, \mathbf{z}^\sigma$, cnext, then $\mathbf{z}^\sigma$ is the cyclic shift left of $\mathbf{z}$ if and only if*

$$\widehat{\mathbf{z}}^\sigma(\mathbf{X}) = \sum_{\mathbf{j} \in \{0,1\}^n} \widehat{\text{cnext}}(\mathbf{X}, \mathbf{j}) \cdot \widehat{\mathbf{z}}(\mathbf{j}). \tag{19}$$

**Proof.** For any $0 \le i, j < N$, let $\mathbf{i} = \textbf{toBinary}(i)$ and $\mathbf{j} = \textbf{toBinary}(j)$. We have

$$\begin{aligned}\sum_{\mathbf{j} \in \{0,1\}^n} \text{cnext}(\mathbf{i}, \mathbf{j}) \cdot \mathbf{z}(\mathbf{j}) &= \text{cnext}(\mathbf{i}, \textbf{toBinary}((i+1) \mod N)) \cdot \mathbf{z}(\textbf{toBinary}((i+1) \mod N)) \\ &= \mathbf{z}(\textbf{toBinary}((i+1) \mod N)) = z_{(i+1) \mod N}.\end{aligned}$$

If $\mathbf{z}^\sigma$ is the cyclic shift left of $\mathbf{z}$, then

$$\mathbf{z}^\sigma(\mathbf{i}) = z_i^\sigma = z_{(i+1) \mod N} = \sum_{\mathbf{j} \in \{0,1\}^n} \text{cnext}(\mathbf{i}, \mathbf{j}) \cdot \mathbf{z}(\mathbf{j}),$$

the Equation 18 holds. Similarly, if the Equation 18 holds, then

$$z_i^\sigma = \mathbf{z}^\sigma(\mathbf{i}) = \sum_{\mathbf{j} \in \{0,1\}^n} \text{cnext}(\mathbf{i}, \mathbf{j}) \cdot \mathbf{z}(\mathbf{j}) = z_{(i+1) \mod N},$$

$\mathbf{z}^\sigma$ is the cyclic shift left of $\mathbf{z}$.

The sencond part of the lemma holds because the left side and right side of Equation 19 are multilinear polynomials and the multilinear extension is unique. □

**Lemma 4.16.** *Let $\widehat{\text{cnext}} : \mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}$ be the multilinear extension of* cnext, *then for any $(\mathbf{r}, \mathbf{r}') \in \mathbb{F}^n \times \mathbb{F}^n$, $\widehat{\text{cnext}}(\mathbf{r}, \mathbf{r}')$ can be computed with $O(n) = O(\log N)$ field operations.*

**Proof.** Let next : $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be a function with the same functionality as next, except that $\text{next}(\mathbf{1}, \mathbf{j}) = 0$ for any $\mathbf{j} \in \{0,1\}^n$. Let $\widehat{\text{next}}$ be the multilinear extension of next, Setty et al. [21] provides the explicit expression for the function $\widehat{\text{next}}$.

$$\widehat{\text{next}}(X_1, \cdots, X_n, Y_1, \cdots, Y_n) = h(X_1, \cdots, X_n, Y_1, \cdots, Y_n) + g(X_1, \cdots, X_n, Y_1, \cdots, Y_n),$$

where

$$h(X_1,\cdots,X_n,Y_1,\cdots,Y_n) = (1-X_n)\cdot Y_n \cdot \widehat{\mathrm{eq}}(X_1,\cdots,X_{n-1},Y_1,\cdots,Y_{n-1})$$

and

$$g(X_1,\cdots,X_n,Y_1,\cdots,Y_n)$$

equals

$$\sum_{k=1}^{n-1}\left(\prod_{i=0}^{k-1} X_{n-i}(1-Y_{n-i})\right)\cdot(1-X_{n-k})\cdot Y_{n-k}\cdot \widehat{\mathrm{eq}}(X_1,\cdots,X_{n-k-1},Y_1,\cdots,Y_{n-k-1}).$$

From $\widehat{\mathrm{next}}$, it is easy to construct the multiliear polynomial $\widehat{\mathrm{cnext}}$,

$$\widehat{\mathrm{cnext}}(\mathbf{X},\mathbf{Y}) = \widehat{\mathrm{next}}(\mathbf{X},\mathbf{Y}) + X_1\cdot X_2\cdots X_n\cdot(1-Y_1)\cdot(1-Y_2)\cdots(1-Y_n).$$

For any $\mathbf{i},\mathbf{j}\in\{0,1\}^n$, if $\mathbf{i}\neq\mathbf{1}$, then $i_1\cdot i_2\cdots i_n=0$ and $\widehat{\mathrm{cnext}}(\mathbf{i},\mathbf{j})=\widehat{\mathrm{next}}(\mathbf{i},\mathbf{j})$. If $\mathbf{i}=\mathbf{1}$, then $\mathrm{next}(\mathbf{1},\mathbf{j})=0$ for any $\mathbf{j}$, $\widehat{\mathrm{cnext}}(\mathbf{1},\mathbf{j})=1$ if and only if $(1-j_1)\cdots(1-j_n)=1$, which means that $\mathbf{j}=\mathbf{0}$. Since $\widehat{\mathrm{next}}(\mathbf{r},\mathbf{r}')$ can be evaluated in $O(n)=O(\log N)$ field operations, the same fact also holds for $\widehat{\mathrm{cnext}}(\mathbf{r},\mathbf{r}')$ □

**Protocol 3.** Cyclic shift left protocol:
*Inputs.*

- Prover's input:
  - instance $\mathbb{x}=\big(\mathbb{F},N,[[\widehat{\mathbf{z}}]],[[\widehat{\mathbf{z}}^{\sigma}]]\big)$,
  - witness $\mathbb{w}=(\mathbf{z},\mathbf{z}^{\sigma})$.
- Verifier's input:
  - instance $\mathbb{x}=\big(\mathbb{F},N,[[\widehat{\mathbf{z}}]],[[\widehat{\mathbf{z}}^{\sigma}]]\big)$.

*Goal.*

- The prover convinces the verifer that $\mathbf{z}^{\sigma}$ is the cyclic shift left vector of $\mathbf{z}$.

*The protocol:*

1. Let $n=\log N$, the verifier samples $\boldsymbol{\rho}=(\rho_1,\cdots,\rho_n)\in\mathbb{F}^n$ and queries $[[\widehat{\mathbf{z}}^{\sigma}]]$ at $\boldsymbol{\rho}$, suppose the returned value is $\gamma$. The verifier then sends $\boldsymbol{\rho}$ to the prover.
2. The prover constructs a vector $\boldsymbol{a_\rho}\in\mathbb{F}^N$, for all $\mathbf{v}\in\{0,1\}^n$:
$$\boldsymbol{a_\rho}(\mathbf{v}) = \widehat{\mathrm{cnext}}(\boldsymbol{\rho},\mathbf{v}).$$
3. Let
   - $\mathbb{x}'=\big(\mathbb{F},N,t,d,F,[[\widehat{\boldsymbol{a}}_{\boldsymbol{\rho}}]],[[\widehat{\mathbf{z}}]],\gamma\big)$,
   - $\mathbb{w}'=\big(\boldsymbol{a_\rho},\mathbf{z}\big)$,

   where $t=2$, $d=2$ and $F(X_1,X_2)=X_1\cdot X_2$. The prover and verifier invoke the SumCheck protocol for checking that $(\mathbb{x}',\mathbb{w}')\in\mathcal{R}_{SUM}$.

Note that the verifer has virtual oracle access to $[[\widehat{\mathbf{a}}_{\boldsymbol{\rho}}]]$, since $\widehat{\mathbf{a}}_{\boldsymbol{\rho}}(\mathbf{X})=\widehat{\mathrm{cnext}}(\boldsymbol{\rho},\mathbf{X})$.

**Lemma 4.17.** *The Protocol 3 has perfect completeness.*

**Proof.** Suppose that $\widehat{\mathbf{z}}^{\sigma}$ is the shift left vector of $\mathbf{z}$, then from Lemma 4.15, it follows that

$$\widehat{\mathbf{z}}^{\sigma}(\mathbf{X}) = \sum_{\mathbf{j}\in\{0,1\}^n} \widehat{\mathrm{cnext}}(\mathbf{X},\mathbf{j}) \cdot \widehat{\mathbf{z}}(\mathbf{j}).$$

Suppose the verifier queries the oracle $[[\widehat{\mathbf{z}}^{\sigma}]]$ with $\boldsymbol{\rho}$ and the returned value is $\gamma$ then $\widehat{\mathbf{z}}^{\sigma}(\boldsymbol{\rho}) = \gamma$. Suppose $\boldsymbol{a}_{\rho} \in \mathbb{F}^N$ is a vector such that for all $\mathbf{v} \in \{0,1\}^n$, $\boldsymbol{a}_{\rho}(\mathbf{v}) = \widehat{\mathrm{cnext}}(\boldsymbol{\rho}, \mathbf{v})$, then

$$\begin{aligned}\sum_{\mathbf{j}\in\{0,1\}^n} F\left(\widehat{\mathbf{a}}_{\rho}(\mathbf{j}), \widehat{\mathbf{z}}(\mathbf{j})\right) &= \sum_{\mathbf{j}\in\{0,1\}^n} \widehat{\mathbf{a}}_{\rho}(\mathbf{j}) \cdot \widehat{\mathbf{z}}(\mathbf{j}) \\ &= \sum_{\mathbf{j}\in\{0,1\}^n} \widehat{\mathrm{cnext}}(\boldsymbol{\rho},\mathbf{j}) \cdot \widehat{\mathbf{z}}(\mathbf{j}) \\ &= \widehat{\mathbf{z}}^{\sigma}(\rho) = \gamma.\end{aligned}$$

It follows that $(\mathbb{x}', \mathbb{w}') \in \mathcal{R}_{SUM}$. The completness of this protocol then follows from the completeness of SumCheck protocol. □

**Lemma 4.18.** *The Protocol 3 has soundness error* $O\left(\frac{\log N}{|\mathbb{F}|}\right)$.

**Proof.** Suppose that $\mathbf{z}^{\sigma}$ is not the shift left vector of $\mathbf{z}$, then from Lemma 4.15, it follows that

$$\widehat{\mathbf{z}}^{\sigma}(\mathbf{X}) \neq \sum_{\mathbf{j}\in\{0,1\}^n} \widehat{\mathrm{cnext}}(\mathbf{X},\mathbf{j}) \cdot \widehat{\mathbf{z}}(\mathbf{j}).$$

For a vector $\boldsymbol{\rho} \in \mathbb{F}^n$, define a event $E$ such that

$$\widehat{\mathbf{z}}^{\sigma}(\boldsymbol{\rho}) = \sum_{\mathbf{j}\in\{0,1\}^n} \widehat{\mathrm{cnext}}(\boldsymbol{\rho},\mathbf{j}) \cdot \widehat{\mathbf{z}}(\mathbf{j}),$$

then from Schwartz-Zipple lemma, $\mathbf{Pr}_{\rho\leftarrow\mathbb{F}^n}[E] \le \frac{n}{|\mathbb{F}|} = \frac{\log N}{|\mathbb{F}|}$. Suppose the verifier queries the oracle $[[\widehat{\mathbf{z}}^{\sigma}]]$ and the returned value is $\gamma$ then $\widehat{\mathbf{z}}^{\sigma}(\boldsymbol{\rho}) = \gamma$, if the event $E$ does not happen, it follows that

$$\sum_{\mathbf{j}\in\{0,1\}^n} F\left(\widehat{\mathbf{a}}_{\rho}(\mathbf{j}), \widehat{\mathbf{z}}(\mathbf{j})\right) = \sum_{\mathbf{j}\in\{0,1\}^n} \widehat{\mathrm{cnext}}(\boldsymbol{\rho},\mathbf{j}) \cdot \widehat{\mathbf{z}}(\mathbf{j}) \neq \widehat{\mathbf{z}}^{\sigma}(\boldsymbol{\rho}) = \gamma.$$

It follows that $(\mathbb{x}', \mathbb{w}') \notin \mathcal{R}_{GSP}$. From the soundness of SumCheck protocol, the verifier will accept with probability at most $O\left(\frac{\log N}{|\mathbb{F}|}\right)$ if the event $E$ does not happen. Therefore,

$$\begin{aligned}\mathbf{Pr}\,[\text{Verifier accepts}] &\le \mathbf{Pr}\left[\text{Verifier accepts}\middle|\overline{E}\right] + \mathbf{Pr}\,[E] \\ &\le O\left(\frac{\log N}{|\mathbb{F}|}\right).\end{aligned}$$

□

**Lemma 4.19.** *In the Protocol 3, the round complexity is $O(\log N)$, message complexity is $O(\log N)$, query complexity is $O(1)$ and the verifier has arithmetic complexity $O(\log N)$.*

**Proof.** The verifier first sends $\log N$ random elements to the prover. The overhead in the latter part of the protocol is all due to the SumCheck protocol. This lemma follows from Theorem 4.1. The verifier can computes $\widehat{\boldsymbol{a}}_{\rho}(\mathbf{r}) = \widehat{\mathrm{cnext}}(\boldsymbol{\rho}, \mathbf{r})$ using $O(\log N)$ field operations from Lemma 4.16. □

**Lemma 4.20.** *In the Protocol 3, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$ in the random access model.*

**Proof.** Let $\boldsymbol{T_\rho} = (1-\rho_1, \rho_1) \otimes (1-\rho_2, \rho_2) \otimes \cdots \otimes (1-\rho_n, \rho_n)$, it is not hard to check that the vector $\boldsymbol{a_\rho}$ is the cyclic shift right of $\boldsymbol{T_\rho}$. For any $0 \leq v < N$, let $\mathbf{v} = \mathbf{toBinary}(v)$,

$$\begin{aligned} a_{\rho,v} = \boldsymbol{a_\rho}(\mathbf{v}) = \widehat{\text{cnext}}(\boldsymbol{\rho}, \mathbf{v}) &= \sum_{\mathbf{i},\mathbf{j} \in \{0,1\}^n} \text{cnext}(\mathbf{i}, \mathbf{j}) \cdot \widehat{\text{eq}}(\mathbf{i}, \boldsymbol{\rho}) \cdot \widehat{\text{eq}}(\mathbf{j}, \mathbf{v}) \\ &= \sum_{\mathbf{i} \in \{0,1\}^n} \text{cnext}(\mathbf{i}, \mathbf{v}) \cdot \widehat{\text{eq}}(\mathbf{i}, \boldsymbol{\rho}) \\ &= \widehat{\text{eq}}(\mathbf{toBinary}(v-1), \boldsymbol{\rho}) = \boldsymbol{T_\rho}(\mathbf{toBinary}(v-1)) \\ &= T_{\rho,v-1}, \end{aligned}$$

we assume that $T_{\rho,-1} = T_{\rho,N-1}$, it follows that $\boldsymbol{a_\rho} = (T_{\rho,N-1}, T_{\rho,0}, \cdots, T_{\rho,N-2})$. Since $\boldsymbol{T_\rho}$ can be constructed using $O(N)$ field operation and $O(N)$ space, the vector $\boldsymbol{a_\rho}$ can also be constructed using $O(N)$ field operations and $O(N)$ space. From Theorem 4.1, the prover has aritmetic complexity $O(N)$ and space complexity $O(N)$. □

**Lemma 4.21.** *In the Protocol 3, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over all input streams in the streaming model.*

**Proof.** In the streaming model, the prover can't construct the vector $\boldsymbol{a_\rho}$ explicitly, since it takes large space to store. Recall that in $l$-th round of the SumCheck protocol, the prover should compute $\boldsymbol{a_\rho}^{(l-1)}(\mathbf{v})$ for all $\mathbf{v} \in \{0,1\}^{n-l+1}$, each element takes $O(\log N)$ field operations to construct since

$$\boldsymbol{a_\rho}^{(l-1)}(\mathbf{v}) = \widehat{\text{cnext}}(\boldsymbol{\rho}, v_1, \cdots, v_{n-l+1}, r_{n-l+2}, \cdots, r_n). \tag{20}$$

The element $\mathbf{z}^{(l-1)}(\mathbf{v})$ takes $O(2^{l-1})$ field operations to construct from the input stream $\mathcal{S}(\mathbf{z})$. It follows that in each round of the SumCheck protocol, the arithmetic complexity is $2^{n-l} \cdot O(\log N + 2^{l-1}) = O(N \cdot \log N/2^l + N)$. Therefore, the total arithmetic complexity is $\sum_{l=1}^n O(N \cdot \log N/2^l + N) = O(N \log N)$. The space complexity remains $O(\log N)$. □

## 4.4 Product Check PIOP

**Definition 4.6.** The **rational product** relation $\mathcal{R}_{RPROD}$ is the set of tuples

$$(\mathbb{x}, \mathbb{w}) = \left( (\mathbb{F}, N, [[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]]), (\mathbf{z}_1, \mathbf{z}_2) \right),$$

where

- $N = 2^n$ for some integer $n$,
- $\mathbf{z}_1, \mathbf{z}_2 \in \mathbb{F}^N$,
- $[[\widehat{\mathbf{z}}_1]]$ and $[[\widehat{\mathbf{z}}_2]]$ are a polynomial oracles which compute $\widehat{\mathbf{z}}_1$ and $\widehat{\mathbf{z}}_2$, the multilinear extensions of $\mathbf{z}_1$ and $\mathbf{z}_2$.
- $\prod_{\mathbf{v} \in \{0,1\}^n} \frac{\mathbf{z}_1(\mathbf{v})}{\mathbf{z}_2(\mathbf{v})} = 1$.

**Theorem 4.22.** *There is a polynomial IOP for relation $\mathcal{R}_{PROD}$ with the following properties:*

*Moreover,*

- *in the **random access model**, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$,*

**Table 5:** Properties of product check protocol

| round complexity | message complexity | query complexity | soundness error | verifier's time |
|---|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(1)$ | $O\left(\frac{\log N}{\lvert\mathbb{F}\rvert}\right)$ | $O(\log N)$ |

- *in the* ***streaming model****, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over the input streams.*

**Protocol 4.** Product Check protocol:

*Inputs.*

- Prover's input:
  - instance $\mathbb{x} = \left(\mathbb{F}, N, [[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]]\right)$,
  - witness $\mathbb{w} = (\mathbf{z}_1, \mathbf{z}_2)$.
- Verifier's input:
  - instance $\mathbb{x} = \left(\mathbb{F}, N, [[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]]\right)$.

*Goal.*

- The prover convinces the verifer that $\prod_{\mathbf{v}\in\{0,1\}^n} \frac{\mathbf{z}_1(\mathbf{v})}{\mathbf{z}_2(\mathbf{v})} = 1$.

*The protocol:*

1. Let $n = \log N$, the prover constructs a vector $\mathbf{z} \in \mathbb{F}^N$, such that for all $\mathbf{v} \in \{0,1\}^n$,
$$\mathbf{z}(\mathbf{v}) = \frac{\mathbf{z}_1(\mathbf{v})}{\mathbf{z}_2(\mathbf{v})}.$$
The prover sends the polynomial oracle $[[\widehat{\mathbf{z}}]]$ to the verifier.
2. Let $n = \log N$, the prover constructs a vector $\mathbf{a} \in \mathbb{F}^N$, such that $a_0 = 1$ and for all $0 < v < N$,
$$a_v = \prod_{j=0}^{v-1} z_j,$$
i.e., $\mathbf{a} = \left(1, z_0, z_0 \cdot z_1, \cdots, \prod_{j=0}^{N-2} z_j\right)$. Let $\mathbf{a}^\sigma$ be the cyclic shift left of $\mathbf{a}$. The prover sends the polynomial oracles $[[\widehat{\mathbf{a}}]]$ and $[[\widehat{\mathbf{a}}^\sigma]]$ to the verifier.
3. The verifer queries $[[\widehat{\mathbf{a}}]]$ at $(0, \cdots 0)$, and rejects if the returned value is not 1.
4. Let
   - $\mathbb{x}_1 = \left(\mathbb{F}, N, [[\widehat{\mathbf{a}}]], [[\widehat{\mathbf{a}}^\sigma]]\right)$,
   - $\mathbb{w}_1 = (\mathbf{a}, \mathbf{a}^\sigma)$.

   The prover and verifier invoke the cyclic shift left protocol for checking that $(\mathbb{x}_1, \mathbb{w}_1) \in \mathcal{R}_{CSL}$.
5. Let $\mathbf{a}_1 = \mathbf{z}||\mathbf{z}$, $\mathbf{a}_2 = \mathbf{z}_2||\mathbf{a}$, $\mathbf{a}_3 = \mathbf{z}_1||\mathbf{a}^\sigma$, and let
   - $\mathbb{x}_2 = \left(\mathbb{F}, 2N, t, d, F, [[\widehat{\mathbf{a}}_1]], [[\widehat{\mathbf{a}}_2]], [[\widehat{\mathbf{a}}_3]]\right)$,
   - $\mathbb{w}_2 = (\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3)$,

   where $t = 3$, $d = 2$ and $F(X_1, X_2, X_3) = X_1 \cdot X_2 - X_3$. The prover and verifier invoke the ZeroCheck protocol for checking that $(\mathbb{x}_2, \mathbb{w}_2) \in \mathcal{R}_{ZERO}$.

Note that the verifer has virtual oracle access to $[[\widehat{\mathbf{a}}_1]], [[\widehat{\mathbf{a}}_2]], [[\widehat{\mathbf{a}}_3]]$, since if $\mathbf{a} = \mathbf{b}||\mathbf{c}$, then

$$\widehat{\mathbf{a}}(X_1, X_2, \cdots, X_{n+1}) = (1 - X_1) \cdot \widehat{\mathbf{b}}(X_2, \cdots, X_{n+1}) + X_1 \cdot \widehat{\mathbf{c}}(X_2, \cdots, X_{n+1}).$$

**Lemma 4.23.** *The Protocol 4 has perfect completeness.*

**Proof.** Suppose that $\prod_{\mathbf{v}\in\{0,1\}^n} \frac{\mathbf{z}_1(\mathbf{v})}{\mathbf{z}_2(\mathbf{v})} = 1$, if $\mathbf{z}$, $\mathbf{a}$ and $\mathbf{a}^{\sigma}$ are generated honestly, then

$$\mathbf{a} = \left(1, z_0, z_0 \cdot z_1, \cdots, \prod_{j=0}^{N-2} z_j\right),$$

$$\mathbf{a}^{\sigma} = \left(z_0, z_0 \cdot z_1, \cdots, \prod_{j=0}^{N-2} z_j, 1\right).$$

It follows that $\mathbf{z}\circ\mathbf{z}_2 = \mathbf{z}_1$, $\mathbf{z}\circ\mathbf{a} = \mathbf{a}^{\sigma}$ and $\mathbf{a}_1 \circ \mathbf{a}_2 - \mathbf{a}_3 = \mathbf{0}$, therefore, $(\mathbb{x}_1, \mathbb{w}_1) \in \mathcal{R}_{CSL}$ and $(\mathbb{x}_2, \mathbb{w}_2) \in \mathcal{R}_{ZERO}$. The completeness follows from the completeness of cyclic shift left protocol and ZeroCheck protocol. □

**Lemma 4.24.** *The Protocol 4 has soundness error* $O\left(\frac{\log N}{|\mathbb{F}|}\right)$.

**Proof.** Assuming verifier receives polynomial oracles $[[\widehat{\mathbf{z}}^*]]$, $[[\widehat{\mathbf{a}}^*]]$ and $[[\widehat{\mathbf{a}}^{\sigma^*}]]$ in step 1 and 2, where

$$\mathbf{z}^* = \left(z_0^*, z_1^*, \cdots, z_{N-1}^*\right)$$
$$\mathbf{a}^* = \left(1, a_1^*, \cdots, a_{N-1}^*\right)$$
$$\mathbf{a}^{\sigma^*} = \left(a_0^{\sigma^*}, a_1^{\sigma^*}, \cdots, a_{N-1}^{\sigma^*}\right)$$

are generated arbitray by the prover (the first element of $a^*$ is always 1, otherwise the verifier will reject in step 3). We will prove that if the following conditions hold, then $\prod_{\mathbf{v}\in\{0,1\}^n} \frac{\mathbf{z}_1(\mathbf{v})}{\mathbf{z}_2(\mathbf{v})} = 1$:

- $\mathbf{z}^* \circ \mathbf{z}_2 = \mathbf{z}_1$,
- $\mathbf{z}^* \circ \mathbf{a}^* = \mathbf{a}^{\sigma^*}$,
- $\mathbf{a}^{\sigma^*}$ is the cyclic shift left vector of $\mathbf{a}^*$.

From $\mathbf{z}^* \circ \mathbf{a}^* = \mathbf{a}^{\sigma^*}$, we can deduce that

$$\mathbf{a}^{\sigma^*} = \left(z_0^*, z_1^* \cdot a_1^*, \cdots, z_{N-1}^* \cdot a_{N-1}^*\right).$$

From $\mathbf{a}^{\sigma^*}$ is the cyclic shift left vector of $\mathbf{a}^*$, we can deduce that

$$\mathbf{a}^{\sigma^*} = \left(a_1^*, a_2^* \cdots, a_{N-1}^*, 1\right).$$

It follows that $z_0^* = a_1^*$ and for all $0 < j < N-1$, $z_j^* \cdot a_j^* = a_{j+1}^*$ and $z_{N-1}^* \cdot a_{N-1}^* = 1$, we can deduce that $\prod_{\mathbf{v}\in\{0,1\}^n} \mathbf{z}^*(\mathbf{v}) = 1$. Finally, from $\mathbf{z}^* \circ \mathbf{z}_2 = \mathbf{z}_1$, we know that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{z}^*(\mathbf{v}) = \frac{\mathbf{z}_1(\mathbf{v})}{\mathbf{z}_2(\mathbf{v})}$, it follows that $\prod_{\mathbf{v}\in\{0,1\}^n} \frac{\mathbf{z}_1(\mathbf{v})}{\mathbf{z}_2(\mathbf{v})} = 1$.

In order to analyze the soundness error, we assume that $\prod_{\mathbf{v}\in\{0,1\}^n} \frac{\mathbf{z}_1(\mathbf{v})}{\mathbf{z}_2(\mathbf{v})} \neq 1$. From the above arguments, we know one of the following conditions must be violated:

- $\mathbf{a}_1 \circ \mathbf{a}_2 \neq \mathbf{a}_3$, where $\mathbf{a}_1 = \mathbf{z}^*||\mathbf{z}^*$, $\mathbf{a}_2 = \mathbf{z}_2||\mathbf{a}^*$, $\mathbf{a}_3 = \mathbf{z}_1||\mathbf{a}^{\sigma^*}$, in this case $(\mathbb{x}_2, \mathbb{w}_2) \notin \mathcal{R}_{ZERO}$.
- $\mathbf{a}^{\sigma^*}$ is not the cyclic shift left vector of $\mathbf{a}^*$, in this case $(\mathbb{x}_1, \mathbb{w}_1) \notin \mathcal{R}_{CSL}$.

In both case the verifier will accept with probability at most $O\left(\frac{\log N}{|\mathbb{F}|}\right)$. □

**Lemma 4.25.** *In the Protocol 4, the round complexity is $O(\log N)$, message complexity is $O(\log N)$, query complexity is $O(1)$ and the verifier has arithmetic complexity $O(\log N)$.*

**Proof.** This lemma follows from Theorem 4.8 and Theorem 4.14. □

**Lemma 4.26.** *In the Protocol 4, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$ in the random access model.*

**Proof.** The vector $\mathbf{z}$ and vector $\mathbf{a} = \left(1, z_0, z_0 \cdot z_1, \cdots, \prod_{j=0}^{N-2} z_j\right)$ can be constructed using $O(N)$ field operation. Once vector $\mathbf{a}$ is constructed, vectors $\mathbf{a}^\sigma$ is also constructed. From Theorem 4.8 and Theorem 4.14, the arithmetic complexity of prover is $O(N)$ and space complexity is $O(N)$. □

**Lemma 4.27.** *In the Protocol 4, the prover has aritmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over the input streams in the streaming model.*

**Proof.** The streams $\mathcal{S}(\mathbf{z})$, $\mathcal{S}(\mathbf{a})$, $\mathcal{S}(\mathbf{a}^\sigma)$, $\mathcal{S}(\mathbf{a}_1)$, $\mathcal{S}(\mathbf{a}_2)$ and $\mathcal{S}(\mathbf{a}_3)$ can all be constructed directly from the input streams $\mathcal{S}(\mathbf{z}_1)$ and $\mathcal{S}(\mathbf{z}_2)$ with only constant passes to the input streams. The prover for ZeroCheck protocol and cyclic shift left protocol can use these stream to complete the task. Note that the input streams can be accessed through more than one session (but no more than a logarithmic number of sessions).

From Theorem 4.14, the prover makes $O(\log N)$ passes over the stream $\mathcal{S}(\mathbf{a})$ and $\mathcal{S}(\mathbf{a}^\sigma)$ and one pass over these streams only requires one pass over $\mathcal{S}(\mathbf{z}_1)$ and $\mathcal{S}(\mathbf{z}_2)$, so in step 4, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over the input streams $\mathcal{S}(\mathbf{z}_1)$ and $\mathcal{S}(\mathbf{z}_2)$.

Similarly, from Theorem 4.8, the prover makes $O(\log N)$ passes over the streams $\mathcal{S}(\mathbf{a}_1)$, $\mathcal{S}(\mathbf{a}_2)$, $\mathcal{S}(\mathbf{a}_3)$ and one pass over these streams only requires a constant passes over $\mathcal{S}(\mathbf{z}_1)$ and $\mathcal{S}(\mathbf{z}_2)$, so in step 5, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over the input streams $\mathcal{S}(\mathbf{z}_1)$ and $\mathcal{S}(\mathbf{z}_2)$.

Therefore, in total, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over the input streams in the streaming model. □

## 4.5 Permutation Check PIOP

**Definition 4.7.** The **permutation** relation $\mathcal{R}_{PERM}$ is the set of tuples

$$(\mathbb{x}, \mathbb{w}) = \left(\left(\mathbb{F}, N, [[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]]\right), (\mathbf{z}_1, \mathbf{z}_2)\right),$$

where

- $N = 2^n$ for some integer $n$,
- $\mathbf{z}_1, \mathbf{z}_2 \in \mathbb{F}^N$,
- $[[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]]$ are polynomial oracles which compute $\widehat{\mathbf{z}}_1$ and $\widehat{\mathbf{z}}_2$, the multilinear extensions of $\mathbf{z}_1$ and $\mathbf{z}_2$.
- there exists a permutation $\boldsymbol{\tau} : \{0,1\}^n \to \{0,1\}^n$ such that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{z}_2(\mathbf{v}) = \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$.

**Theorem 4.28.** *There is a polynomial IOP for relation $\mathcal{R}_{PERM}$ with the following properties:*

**Table 6:** Properties of permutation check protocol

| round complexity | message complexity | query complexity | soundness error | verifier's time |
|---|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(1)$ | $O\left(\frac{N}{\lvert\mathbb{F}\rvert}\right)$ | $O(\log N)$ |

*Moreover,*

- *in the* ***random access model****, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$,*
- *in the* ***streaming model****, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over the input streams $\mathcal{S}(\mathbf{z}_1)$ and $\mathcal{S}(\mathbf{z}_2)$.*

If two vectors are the same under some permutations, then the two monic polynomials with elements from these two vectors as roots must also be equal.

**Lemma 4.29.** *Suppose $N = 2^n$, let $\mathbf{z}_1, \mathbf{z}_2 \in \mathbb{F}^N$, if there exists a permutation $\boldsymbol{\tau} : \{0,1\}^n \to \{0,1\}^n$ such that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{z}_2(\mathbf{v}) = \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$, then*

$$\prod_{\mathbf{v}\in\{0,1\}^n} (X - \mathbf{z}_1(\mathbf{v})) = \prod_{\mathbf{v}\in\{0,1\}^n} (X - \mathbf{z}_2(\mathbf{v})) . \tag{21}$$

**Protocol 5.** Permutation Check protocol:
*Inputs.*

- Prover's input:
  - instance $\mathbb{x} = (\mathbb{F}, N, [[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]])$,
  - witness $\mathbb{w} = (\mathbf{z}_1, \mathbf{z}_2)$.
- Verifier's input:
  - instance $\mathbb{x} = (\mathbb{F}, N, [[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]])$.

*Goal.*

- The prover convinces the verifer that there exists a permutation $\boldsymbol{\tau} : \{0,1\}^n \to \{0,1\}^n$ such that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{z}_2(\mathbf{v}) = \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$.

*The protocol:*

1. The verifier samples a random element $\alpha \leftarrow \mathbb{F}$ and sends it to the prover.
2. Let $\mathbf{z}'_1 = \alpha \cdot \mathbf{1} - \mathbf{z}_1$, $\mathbf{z}'_2 = \alpha \cdot \mathbf{1} - \mathbf{z}_2$, and let
   - instance $\mathbb{x}' = (\mathbb{F}, N, [[\widehat{\mathbf{z}}'_1]], [[\widehat{\mathbf{z}}'_2]])$,
   - witness $\mathbb{w}' = (\mathbf{z}'_1, \mathbf{z}'_2)$.

   The prover and verifier invoke the product check protocol for checking that $(\mathbb{x}', \mathbb{w}') \in \mathcal{R}_{PROD}$.

Note that the verifer has virtual oracle access to $[[\widehat{\mathbf{z}}'_1]], [[\widehat{\mathbf{z}}'_2]]$, since $\widehat{\mathbf{z}}'_1(\mathbf{X}) = \alpha - \widehat{\mathbf{z}}_1(\mathbf{X})$ and $\widehat{\mathbf{z}}'_2(\mathbf{X}) = \alpha - \widehat{\mathbf{z}}_2(\mathbf{X})$.

**Lemma 4.30.** *The Protocol 5 has perfect completeness.*

**Proof.** Directly follows from Lemma 4.29 and the completeness of product check protocol. □

**Lemma 4.31.** *The Protocol 5 has soundness error* $O\left(\frac{N}{|\mathbb{F}|}\right)$.

**Proof.** From Lemma 4.29, if the vector $\mathbf{z}_2$ is not the permutation of $\mathbf{z}_1$, then the Equation 21 does not hold. From Schwartz-Zippel Lemma, for a uniformly random element $\alpha \in \mathbb{F}$, the following equation holds with probability at most $\frac{N}{|\mathbb{F}|}$:

$$\prod_{\mathbf{v}\in\{0,1\}^n} (\alpha - \mathbf{z}_1(\mathbf{v})) = \prod_{\mathbf{v}\in\{0,1\}^n} (\alpha - \mathbf{z}_2(\mathbf{v})) \,. \tag{22}$$

Let $E$ be the event that Equation 22 holds, then from the soundess of product check protocol, it follows that

$$\begin{aligned} \mathbf{Pr}[\text{Verifier accept}] &\le \mathbf{Pr}[\text{Verifier accept}|\bar{E}] + \mathbf{Pr}[E] \\ &\le O\left(\frac{\log N}{|\mathbb{F}|}\right) + \frac{N}{|\mathbb{F}|} \\ &= O\left(\frac{N}{|\mathbb{F}|}\right). \end{aligned} \tag{23}$$

□

**Lemma 4.32.** *In the Protocol 5, the round complexity is* $O(\log N)$*, message complexity is* $O(\log N)$*, query complexity is* $O(1)$ *and the verifier has arithmetic complexity* $O(\log N)$*.*

**Proof.** This lemma follows from Theorem 4.22. □

**Lemma 4.33.** *In the Protocol 5, the prover has arithmetic complexity* $O(N)$ *and space complexity* $O(N)$ *in the random access model.*

**Proof.** This lemma follows from Theorem 4.22. □

**Lemma 4.34.** *In the Protocol 5, the prover has aritmetic complexity* $O(N \log N)$*, space complexity* $O(\log N)$ *and makes* $O(\log N)$ *passes over the input streams* $\mathcal{S}(\mathbf{z}_1)$ *and* $\mathcal{S}(\mathbf{z}_2)$ *in the streaming model.*

**Proof.** This lemma follows from Theorem 4.22 since the streams $\mathcal{S}(\mathbf{z}'_1)$ and $\mathcal{S}(\mathbf{z}'_2)$ can be easily constructed from the streams $\mathcal{S}(\mathbf{z}_1)$ and $\mathcal{S}(\mathbf{z}_2)$ with a single pass. □

## 4.6 Prescribed Permutation Check PIOP

**Definition 4.8.** The indexed **prescribed permutation** relation $\mathcal{R}_{PREP}$ is the set of tuples

$$(\mathbb{i}, \mathbb{x}, \mathbb{w}) = \left((\mathbb{F}, N, \boldsymbol{\phi}, \boldsymbol{\tau}), ([[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]]), (\mathbf{z}_1, \mathbf{z}_2)\right),$$

where

- $N = 2^n$ for some integer $n$,
- $\boldsymbol{\phi} : \{0,1\}^n \to \mathbb{F}$ is a canonical injection from $\{0,1\}^n$ to $\mathbb{F}$. E.g., $\boldsymbol{\phi} =$ **toDecimal** and $\boldsymbol{\phi}^{-1} =$ **toBinary** if $\mathbb{F} = \mathbb{F}_p$ for some $p \ge N$.
- $\mathbf{z}_1, \mathbf{z}_2 \in \mathbb{F}^N$.

- $\boldsymbol{\tau} : \{0,1\}^n \to \{0,1\}^n$ is the permutation on the Boolean hypercube.
- $[[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]]$ are polynomial oracles which compute $\widehat{\mathbf{z}}_1, \widehat{\mathbf{z}}_2$, the multilinear extensions of $\mathbf{z}_1, \mathbf{z}_2$.
- for every $\mathbf{v} \in \{0,1\}^n$, $\mathbf{z}_2(\mathbf{v}) = \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$.

**Theorem 4.35.** *There is a polynomial holographic IOP for relation $\mathcal{R}_{PREP}$ with the following properties:*

**Table 7:** Properties of prescribed permutation check protocol

| round complexity | message complexity | query complexity | soundness error | verifier's time |
|---|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(1)$ | $O\left(\frac{\log N}{\lvert\mathbb{F}\rvert}\right)$ | $O(\log N)$ |

*Moreover,*

- *in the **random access model**, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$,*
- *in the **streaming model**, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over all input streams.*

**Lemma 4.36.** *Suppose $N = 2^n$, fix a canonical injection $\boldsymbol{\phi} : \{0,1\}^n \to \mathbb{F}$, let $\mathbf{z}_1, \mathbf{z}_2 \in \mathbb{F}^N$ and $\boldsymbol{\tau} : \{0,1\}^n \to \{0,1\}^n$ is a permutation on the Boolean hypercube. Let $\boldsymbol{\tau}' \in \mathbb{F}^N$ such that for all $\mathbf{v} \in \{0,1\}^n$, $\boldsymbol{\tau}'(\mathbf{v}) = \boldsymbol{\phi}(\boldsymbol{\tau}(\mathbf{v}))$. For a uniformly random chosen element $\beta \leftarrow \mathbb{F}$, define two vectors $\mathbf{z}'_1, \mathbf{z}'_2 \in \mathbb{F}^N$:*

- $\mathbf{z}'_1 = \mathbf{z}_1 + \beta \cdot \boldsymbol{\phi}$,
- $\mathbf{z}'_2 = \mathbf{z}_2 + \beta \cdot \boldsymbol{\tau}'$.

*If for every $\mathbf{v} \in \{0,1\}^n$, $\mathbf{z}_2(\mathbf{v}) = \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$, then $\mathbf{z}'_2$ is a permutation of $\mathbf{z}'_1$ with probability 1. If there exists a $\mathbf{v} \in \{0,1\}^n$ such that $\mathbf{z}_2(\mathbf{v}) \neq \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$, then $\mathbf{z}'_2$ is a permutation of $\mathbf{z}'_1$ with probability at most $\frac{N}{\lvert\mathbb{F}\rvert}$.*

**Proof.** For any $\beta \leftarrow \mathbb{F}$, suppose for every $\mathbf{v} \in \{0,1\}^n$, $\mathbf{z}_2(\mathbf{v}) = \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$, then

$$\mathbf{z}'_1(\boldsymbol{\tau}(\mathbf{v})) = \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v})) + \beta \cdot \boldsymbol{\phi}(\boldsymbol{\tau}(\mathbf{v})) = \mathbf{z}_2(\mathbf{v}) + \beta \cdot \boldsymbol{\tau}'(\mathbf{v}) = \mathbf{z}'_2(\mathbf{v}). \tag{24}$$

Suppose there exists a $\mathbf{v} \in \{0,1\}^n$ such that $\mathbf{z}_2(\mathbf{v}) \neq \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$, then for any particular $\mathbf{u} \in \{0,1\}^n$, there is at most one $\beta$ such that

$$\mathbf{z}_1(\mathbf{u}) + \beta \cdot \boldsymbol{\phi}(\mathbf{u}) = \mathbf{z}_2(\mathbf{v}) + \beta \cdot \boldsymbol{\tau}'(\mathbf{v}). \tag{25}$$

In other words, the probability that $\mathbf{z}'_1(\mathbf{u}) = \mathbf{z}'_2(\mathbf{v})$ is at most $\frac{1}{\lvert\mathbb{F}\rvert}$. By a union bound, the probability that there exists a $\mathbf{u} \in \{0,1\}^n$ such that $\mathbf{z}'_1(\mathbf{u}) = \mathbf{z}'_2(\mathbf{v})$ is at most $\frac{N}{\lvert\mathbb{F}\rvert}$. It follows that the probability that $\mathbf{z}'_2$ is a permutation of $\mathbf{z}'_1$ is at most $\frac{N}{\lvert\mathbb{F}\rvert}$.

□

**Protocol 6.** Prescribed Permutation Check protocol:
*Inputs.*

- Indexer's input:
  - index $\mathbb{i} = (\mathbb{F}, N, \boldsymbol{\phi}, \boldsymbol{\tau})$.
- Prover's input:

  - instance $\mathbb{x} = ([[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]])$,
  - witness $\mathbb{w} = (\mathbf{z}_1, \mathbf{z}_2)$.
- Verifier's input:
  - instance $\mathbb{x} = ([[\widehat{\mathbf{z}}_1]], [[\widehat{\mathbf{z}}_2]])$.

*Goal.*

- The prover convinces the verifer that for all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{z}_2(\mathbf{v}) = \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$.

*The protocol:*

- *Offline phase:*
  1. The indexer constructs a vector $\boldsymbol{\tau}' \in \mathbb{F}^N$ such that for all $\mathbf{v} \in \{0,1\}^n$, $\boldsymbol{\tau}'(\mathbf{v}) = \boldsymbol{\phi}(\boldsymbol{\tau}(\mathbf{v}))$.
  2. The indexer sends $\mathbb{F}$, $N$, $\boldsymbol{\phi}$ and $\boldsymbol{\tau}'$ to the prover.
  3. The indexer sends $\mathbb{F}$, $N$, $[[\widehat{\boldsymbol{\phi}}]]$ and $[[\widehat{\boldsymbol{\tau}}']]$ to the verifier.
- *Online phase:*
  1. The verifier samples a random element $\beta \leftarrow \mathbb{F}$ and sends it to the prover.
  2. Let $\mathbf{z}_1' = \mathbf{z}_1 + \beta \cdot \boldsymbol{\phi}$, $\mathbf{z}_2' = \mathbf{z}_2 + \beta \cdot \boldsymbol{\tau}'$, and let
     - instance $\mathbb{x}' = (\mathbb{F}, N, [[\widehat{\mathbf{z}}_1']], [[\widehat{\mathbf{z}}_2']])$,
     - witness $\mathbb{w}' = (\mathbf{z}_1', \mathbf{z}_2')$.

     The prover and verifier invoke the permutation check protocol for checking that $(\mathbb{x}', \mathbb{w}') \in \mathcal{R}_{PERM}$.

Note that the verifer has virtual oracle access to $[[\widehat{\mathbf{z}}_1']]$, $[[\widehat{\mathbf{z}}_2']]$, since $\widehat{\mathbf{z}}_1'(\mathbf{X}) = \widehat{\mathbf{z}}_1(\mathbf{X}) + \beta \cdot \widehat{\boldsymbol{\phi}}(\mathbf{X})$, $\widehat{\mathbf{z}}_2'(\mathbf{X}) = \widehat{\mathbf{z}}_2(\mathbf{X}) + \beta \cdot \widehat{\boldsymbol{\tau}}'(\mathbf{X})$.

**Lemma 4.37.** *The Protocol 6 has perfect completeness.*

**Proof.** Directly follows from Lemma 4.36 and the completeness of the permutation check protocol. □

**Lemma 4.38.** *The Protocol 6 has soundness error $O\left(\frac{N}{|\mathbb{F}|}\right)$.*

**Proof.** From Lemma 4.36, if there exists a $\mathbf{v} \in \{0,1\}^n$ such that $\mathbf{z}_2(\mathbf{v}) \neq \mathbf{z}_1(\boldsymbol{\tau}(\mathbf{v}))$, then $\mathbf{z}_2'$ is a permutation of $\mathbf{z}_1'$ with probability at most $\frac{N}{|\mathbb{F}|}$. Let $E$ be the event that $\mathbf{z}_2'$ is a permutation of $\mathbf{z}_1'$, then from the soundess of permutation check protocol, it follows that

$$\begin{aligned} \mathbf{Pr}[\text{Verifier accept}] &\leq \mathbf{Pr}[\text{Verifier accept}|\overline{E}] + \mathbf{Pr}[E] \\ &\leq O\left(\frac{N}{|\mathbb{F}|}\right) + \frac{N}{|\mathbb{F}|} \\ &= O\left(\frac{N}{|\mathbb{F}|}\right). \end{aligned} \tag{26}$$

□

**Lemma 4.39.** *In the Protocol 6, the round complexity is $O(\log N)$, message complexity is $O(\log N)$, query complexity is $O(1)$ and the verifier has arithmetic complexity $O(\log N)$.*

**Proof.** This lemma follows from Theorem 4.28. □

**Lemma 4.40.** *In the Protocol 6, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$ in the random access model.*

**Proof.** This lemma follows from Theorem 4.28. □

**Lemma 4.41.** *In the Protocol 6, the prover has aritmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over the input streams $\mathcal{S}(\mathbf{z}_1)$ and $\mathcal{S}(\mathbf{z}_2)$ in the streaming model.*

**Proof.** In the streaming model, the prover receives the streams $\mathcal{S}(\boldsymbol{\phi})$, $\mathcal{S}(\boldsymbol{\tau}')$ as inputs. This lemma follows from Theorem 4.28 since the streams $\mathcal{S}(\mathbf{z}_1')$ and $\mathcal{S}(\mathbf{z}_2')$ can be easily constructed from the streams $\mathcal{S}(\mathbf{z}_1)$, $\mathcal{S}(\mathbf{z}_2)$, $\mathcal{S}(\boldsymbol{\phi})$ and $\mathcal{S}(\boldsymbol{\tau}')$ with a single pass. □

## 5 Elastic PIOP for Plonk Constraint System

### 5.1 Streaming Plonk Constraint System

We introduce the streaming Plonk constraint system. We begin by recalling the indexed Plonk constraint system relation $\mathcal{R}_{PLONK}$.

**Definition 5.1.** The indexed relation $\mathcal{R}_{PLONK}$ is the set of all triples

$$(\mathbb{i}, \mathbb{x}, \mathbb{w}) = ((\mathbb{F}, N, N_{in}, d, G, \boldsymbol{\phi}, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \boldsymbol{\tau}), (\mathbf{x}), (\mathbf{l}, \mathbf{r}, \mathbf{o})),$$

where

- $N = 2^n$, $N_{in} = 2^{n_{in}}$ for some integer $n$, $n_{in}$, $d$ is a constant.
- $G \in \mathbb{F}[X_1, X_2]$ is a custom gate with degree at most $d$.
- $\boldsymbol{\phi} : \{0,1\}^{2n} \to \mathbb{F}$ is a canonical injection from $\{0,1\}^{2n}$ to $\mathbb{F}$. E.g., $\boldsymbol{\phi} =$ **toDecimal** and $\boldsymbol{\phi}^{-1} =$ **toBinary** if $\mathbb{F} = \mathbb{F}_p$ for some $p \geq 4N$.
- $\mathbf{s}_1, \mathbf{s}_2, \mathbf{l}, \mathbf{r}, \mathbf{o} \in \mathbb{F}^N$, $\mathbf{x} \in \mathbb{F}^{N_{in}}$,
- $\boldsymbol{\tau} : \{0,1\}^{n+2} \to \{0,1\}^{n+2}$ is a permutation on the Boolean hypercube,

such that

- for all $\mathbf{v} \in \{0,1\}^n$,

$$\mathbf{s}_1(\mathbf{v}) \cdot (\mathbf{l}(\mathbf{v}) + \mathbf{r}(\mathbf{v})) + \mathbf{s}_2(\mathbf{v}) \cdot \mathbf{l}(\mathbf{v}) \cdot \mathbf{r}(\mathbf{v}) + \mathbf{s}_3(\mathbf{v}) \cdot G(\mathbf{l}(\mathbf{v}), \mathbf{r}(\mathbf{v})) - \mathbf{o}(\mathbf{v}) + \mathbf{x}'(\mathbf{v}) = 0, \quad (27)$$

  where $\mathbf{x}' = \mathbf{x}||\mathbf{0} \in \mathbb{F}^N$ (padding $\mathbf{x}$ with $N - N_{in}$ zeros),
- for all $\mathbf{v}' \in \{0,1\}^{n+2}$, $\mathbf{w}(\mathbf{v}') = \mathbf{w}(\boldsymbol{\tau}(\mathbf{v}'))$, where $\mathbf{w} = \mathbf{l}||\mathbf{r}||\mathbf{o}||\mathbf{0} \in \mathbb{F}^{4N}$.

The relation $\mathcal{R}_{PLONK}$ captures the fan-in two arithmetic ciruit computations. Every gates in the circuit is indexed by the Boolean hypercube $\{0,1\}^n$, the public input/output gates is indexed by the Boolean hypercube $\{0,1\}^{n_{in}}$ (we assume the input of the circuit is the “ouput” of the input gates and we ignore the “input” of the input gate). For all $\mathbf{v} \in \{0,1\}^n$, $\mathbf{l}(\mathbf{v})$ denotes the value on the left input wire of the $\mathbf{v}$-th gate, $\mathbf{r}(\mathbf{v})$ denotes the value on the right input wire of the $\mathbf{v}$-th gate and $\mathbf{o}(\mathbf{v})$ denotes the value on the output wire of the $\mathbf{v}$-th gate. For all $\mathbf{v} \in \{0,1\}^{n_{in}}$, $\mathbf{x}(\mathbf{v})$ denotes the value on the output wire of the $\mathbf{v}$-th public input/output gate (i.e., the vector $\mathbf{x}$ denotes the public input/output values). The polynomial $G$ represents the custom gate. The vectors $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{s}_3$ is the selector vectors and defined as follows:

- for an addition gate: $\mathbf{s}_1(\mathbf{v}) = 1, \mathbf{s}_2(\mathbf{v}) = 0, \mathbf{s}_3(\mathbf{v}) = 0$,
- for a multiplication gate: $\mathbf{s}_1(\mathbf{v}) = 0, \mathbf{s}_2(\mathbf{v}) = 1, \mathbf{s}_3(\mathbf{v}) = 0$,
- for a custom gate: $\mathbf{s}_1(\mathbf{v}) = 0, \mathbf{s}_2(\mathbf{v}) = 0, \mathbf{s}_3(\mathbf{v}) = 1$,
- for an input gate: $\mathbf{s}_1(\mathbf{v}) = 0, \mathbf{s}_2(\mathbf{v}) = 0, \mathbf{s}_3(\mathbf{v}) = 0$.

It is not hard to check that if all gates were evaluated correctly, then for all $\mathbf{v} \in \{0,1\}^n$, the Equation 27 holds. Another additional thing need to check is that some values in $\mathbf{l}$, $\mathbf{r}$, $\mathbf{o}$ are equal, the so-called copy constraint (the output of one gate is the input of another gate). The copy constraint is determined by the permutation $\boldsymbol{\tau}$, let $\mathbf{w} = \mathbf{l}||\mathbf{r}||\mathbf{o}||\mathbf{0}$ be the all wire values of the circuit (we pad $\mathbf{w}$ with $N$ zeros such that the length of $\mathbf{w}$ is a power of 2), then the copy constraint is satisfied if for all $\mathbf{v}' \in \{0,1\}^{n+2}$, $\mathbf{w}(\mathbf{v}') = \mathbf{w}(\boldsymbol{\tau}(\mathbf{v}'))$.

For streaming Plonk constraint system, all vectors in the $\mathcal{R}_{PLONK}$ need to be provided in the form of streams, we defined as follows:

**Definition 5.2.** The streams associated with $(\mathbb{i}, \mathbb{x}, \mathbb{w}) = ((\mathbb{F}, N, N_{in}, d, G, \boldsymbol{\phi}, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \boldsymbol{\tau}), (\mathbf{x}), (\mathbf{l}, \mathbf{r}, \mathbf{o}))$ consists of:

- **index stream**: stream of the vectors $\mathcal{S}(\boldsymbol{\phi}), \mathcal{S}(\mathbf{s}_1), \mathcal{S}(\mathbf{s}_2), \mathcal{S}(\mathbf{s}_3), \mathcal{S}(\boldsymbol{\tau})$,
- **instance stream**: stream of the vector $\mathcal{S}(\mathbf{x})$,
- **witness stream**: stream of the vectors $\mathcal{S}(\mathbf{l})$, $\mathcal{S}(\mathbf{r})$, $\mathcal{S}(\mathbf{o})$.

*Remark.* Because $\boldsymbol{\phi}$ is a canonical injection, we assume that $\boldsymbol{\phi}$ can be computed with a constant number of field operations. But for convenience, we also assume that $\boldsymbol{\phi}$ is given as a stream input $\mathcal{S}(\boldsymbol{\phi})$. The permutation $\boldsymbol{\tau}$ can also be viewed as a vector $\boldsymbol{\tau} \in \left(\{0,1\}^{n+2}\right)^{4N}$. We can assume $\boldsymbol{\tau}$ is given as a stream $\mathcal{S}(\boldsymbol{\tau})$ in the streaming model.

## 5.2 Construction

**Theorem 5.1.** *There is a polynomial holographic IOP for relation $\mathcal{R}_{PLONK}$ with the following properties:*

**Table 8:** Properties of protocol for Plonk constraint system

| round complexity | message complexity | query complexity | soundness error | verifier's time |
|---|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(1)$ | $O\left(\frac{N}{\lvert\mathbb{F}\rvert}\right)$ | $O(\lvert\mathbf{x}\rvert + \log N)$ |

*Moreover,*

- *in the **random access model**, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$,*
- *in the **streaming model**, the prover has arithmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over all input streams.*

**Protocol 7.** Protocol for Plonk constraint system:
*Inputs.*

- Indexer's input:
  - index $\mathbb{i} = (\mathbb{F}, N, N_{in}, d, G, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \boldsymbol{\tau})$.
- Prover's input:
  - instance $\mathbb{x} = (\mathbf{x})$,
  - witness $\mathbb{w} = (\mathbf{l}, \mathbf{r}, \mathbf{o})$.
- Verifier's input:
  - instance $\mathbb{x} = (\mathbf{x})$.

*Goal.*

- The prover convinces the verifer that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}_{PLONK}$.

*The protocol:*

- *Offline phase:*
  1. The indexer construct a multivariate polynomial $F \in \mathbb{F}[X_1, \cdots X_7]$:
  $$F(X_1, \cdots, X_7) = X_1 \cdot (X_4 + X_5) + X_2 \cdot X_4 \cdot X_5 + X_3 \cdot G(X_4, X_5) - X_6 + X_7, \tag{28}$$
  note that the total degree of the polynomial $F$ is at most $d + 1$.
  2. The indexer constructs a vector $\boldsymbol{\tau}' \in \mathbb{F}^{4N}$ such that for all $\mathbf{v} \in \{0, 1\}^{n+2}$, $\boldsymbol{\tau}'(\mathbf{v}) = \boldsymbol{\phi}(\boldsymbol{\tau}(\mathbf{v}))$.
  3. The indexer sends $\mathbb{F}$, $N$, $N_{in}$, $\boldsymbol{\phi}$, $\boldsymbol{\tau}'$, $\mathbf{s}_1$, $\mathbf{s}_2$, $\mathbf{s}_3$ to the prover.
  4. The indexer sends $\mathbb{F}$, $N$, $N_{in}$, $[[\widehat{\boldsymbol{\phi}}]]$, $[[\widehat{\boldsymbol{\tau}}']]$, $[[\widehat{\mathbf{s}}_1]]$, $[[\widehat{\mathbf{s}}_2]]$, $[[\widehat{\mathbf{s}}_3]]$ to the verifier.
- *Online phase:*
  1. The prover sends polynomial oracles $[[\widehat{\mathbf{l}}]]$, $[[\widehat{\mathbf{r}}]]$, $[[\widehat{\mathbf{o}}]]$ to the verifier.
  2. Let
     - instance $\mathbb{x}_1 = \left(\mathbb{F}, N, 7, d + 1, F, [[\widehat{\mathbf{s}}_1]], [[\widehat{\mathbf{s}}_2]], [[\widehat{\mathbf{s}}_3]], [[\widehat{\mathbf{l}}]], [[\widehat{\mathbf{r}}]], [[\widehat{\mathbf{o}}]], [[\widehat{\mathbf{x}}']]\right)$,
     - witness $\mathbb{w}_1 = (\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{l}, \mathbf{r}, \mathbf{o}, \mathbf{x}')$,

     where $\mathbf{x}' = \mathbf{x}||\mathbf{0} \in \mathbb{F}^N$. The prover and verifier invoke the ZeroCheck protocol for checking that $(\mathbb{x}_1, \mathbb{w}_1) \in \mathcal{R}_{ZERO}$.
  3. Let $\mathbf{w} = \mathbf{l}||\mathbf{r}||\mathbf{o}||\mathbf{0} \in \mathbb{F}^{4N}$, where $\mathbf{0}$ is the all zero vector. Let
     - index $\mathbb{i}_2 = (\mathbb{F}, 4N, \boldsymbol{\phi}, \boldsymbol{\tau})$
     - instance $\mathbb{x}_2 = \left(\mathbb{F}, 4N, [[\widehat{\mathbf{w}}]], [[\widehat{\mathbf{w}}]]\right)$,
     - witness $\mathbb{w}_2 = (\mathbf{w}, \mathbf{w})$.

     The prover and verifier invoke the online phase of prescribed permutation check protocol for checking that $(\mathbb{i}_2, \mathbb{x}_2, \mathbb{w}_2) \in \mathcal{R}_{PREP}$.

Note that the verifer has virtual oracle access to $[[\widehat{\mathbf{x}}']]$ and $[[\widehat{\mathbf{w}}]]$, since
$$\widehat{\mathbf{x}}'(X_1, \cdots, X_n) = \widehat{\mathbf{x}}(X_{n-n_{in}+1}, \cdots, X_n) \cdot (1 - X_1) \cdots (1 - X_{n-n_{in}}), \tag{29}$$
and
$$\widehat{\mathbf{w}}(X_1, X_2, \mathbf{X}) = (1 - X_1) \cdot (1 - X_2) \cdot \widehat{\mathbf{l}}(\mathbf{X}) + (1 - X_1) \cdot X_2 \cdot \widehat{\mathbf{r}}(\mathbf{X}) + X_1 \cdot (1 - X_2) \cdot \widehat{\mathbf{o}}(\mathbf{X}), \tag{30}$$
where $\mathbf{X} = (X_3, \cdots, X_{n+2})$.

**Lemma 5.2.** *The Protocol 7 has perfect completeness.*

**Proof.** From the construction of the polynomial $F$ (Equation 28), we can get for all $\mathbf{v} \in \{0,1\}^n$,

$$\begin{aligned} &F\left(\widehat{\mathbf{s}}_1(\mathbf{v}), \widehat{\mathbf{s}}_2(\mathbf{v}), \widehat{\mathbf{s}}_3(\mathbf{v}), \widehat{\mathbf{l}}(\mathbf{v}), \widehat{\mathbf{r}}(\mathbf{v}), \widehat{\mathbf{o}}(\mathbf{v}), \widehat{\mathbf{x}}'(\mathbf{v})\right) \\ &\quad = \widehat{\mathbf{s}}_1(\mathbf{v}) \cdot \left(\widehat{\mathbf{l}}(\mathbf{v}) + \widehat{\mathbf{r}}(\mathbf{v})\right) + \widehat{\mathbf{s}}_2(\mathbf{v}) \cdot \widehat{\mathbf{l}}(\mathbf{v}) \cdot \widehat{\mathbf{r}}(\mathbf{v}) + \widehat{\mathbf{s}}_3(\mathbf{v}) \cdot G\left(\widehat{\mathbf{l}}(\mathbf{v}), \widehat{\mathbf{r}}(\mathbf{v})\right) - \widehat{\mathbf{o}}(\mathbf{v}) + \widehat{\mathbf{x}}'(\mathbf{v}) \\ &\quad = \mathbf{s}_1(\mathbf{v}) \cdot (\mathbf{l}(\mathbf{v}) + \mathbf{r}(\mathbf{v})) + \mathbf{s}_2(\mathbf{v}) \cdot \mathbf{l}(\mathbf{v}) \cdot \mathbf{r}(\mathbf{v}) + \mathbf{s}_3(\mathbf{v}) \cdot G(\mathbf{l}(\mathbf{v}), \mathbf{r}(\mathbf{v})) - \mathbf{o}(\mathbf{v}) + \mathbf{x}'(\mathbf{v}) = 0. \end{aligned} \tag{31}$$

Therefore, $(\mathbb{i}_1, \mathbb{w}_2) \in \mathcal{R}_{ZERO}$. Since for all $\mathbf{v}' \in \{0,1\}^{n+2}$, $\mathbf{w}(\mathbf{v}') = \mathbf{w}(\boldsymbol{\tau}(\mathbf{v}'))$, it follows that $(\mathbb{i}_2, \mathbb{x}_2, \mathbb{w}_2) \in \mathcal{R}_{PREP}$. The completeness follows from the completeness of ZeroCheck and prescribed permutation check protocol. □

**Lemma 5.3.** *The Protocol 7 has soundness error $O\left(\frac{N}{|\mathbb{F}|}\right)$.*

**Proof.** If $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R}_{PLONK}$, then one of the following conditions must be violated:

- $(\mathbb{x}_1, \mathbb{w}_1) \notin \mathcal{R}_{ZERO}$,
- $(\mathbb{i}_2, \mathbb{x}_2, \mathbb{w}_2) \notin \mathcal{R}_{PREP}$.

In the first case, from the soundenss of ZeroCheck protocol, the verifier will accept with probability at most $O\left(\frac{\log N}{|\mathbb{F}|}\right)$. In the second case, from the soundenss of prescribed permutation check protocol, the verifier will accept with probability at most $O\left(\frac{N}{|\mathbb{F}|}\right)$. In summary, for any $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R}_{PLONK}$, the verifier will accept with probability at most $\max\left\{O\left(\frac{\log N}{|\mathbb{F}|}\right), O\left(\frac{N}{|\mathbb{F}|}\right)\right\} = O\left(\frac{N}{|\mathbb{F}|}\right)$. □

**Lemma 5.4.** *In the Protocol 7, the round complexity is $O(\log N)$, message complexity is $O(\log N)$, query complexity is $O(1)$ and the verifier has arithmetic complexity $O(|\mathbf{x}| + \log N)$.*

**Proof.** The verifier can evaluates $\widehat{\mathbf{x}}$ at any points with $O(|\mathbf{x}|)$ field operations. This lemma then follows from Theorem 4.8 and Theorem 4.35. □

**Lemma 5.5.** *In the Protocol 6, the prover has arithmetic complexity $O(N)$ and space complexity $O(N)$ in the random access model.*

**Proof.** This lemma follows from Theorem 4.8 and Theorem 4.35. □

**Lemma 5.6.** *In the Protocol 6, the prover has aritmetic complexity $O(N \log N)$, space complexity $O(\log N)$ and makes $O(\log N)$ passes over the input streams in the streaming model.*

**Proof.** In the streaming model, the prover receives the streams $\mathcal{S}(\boldsymbol{\phi})$, $\mathcal{S}(\boldsymbol{\tau}')$, $\mathcal{S}(\mathbf{s}_1)$, $\mathcal{S}(\mathbf{s}_2)$, $\mathcal{S}(\mathbf{s}_3)$ as inputs. This lemma follows from Theorem 4.8 and Theorem 4.35, since all the streams for the ZeroCheck protocol and prescribed permutation check protocol can be constructed from the input streams with a constant number of passes. □

## 6 Elastic KZG Scheme for Multilinear Polynomials

Similar to the construction of most succinct arguments, we utilize a polynomial commitment scheme to compile our polynomial IOP into a succinct argument. To achieve elastic arguments, we also require the polynomial commitment scheme to be elastic. Gemini proposed an elastic univariate polynomial commitment scheme based on the KZG scheme [12]. Based on their construction, we found that the multilinear KZG scheme [14] can also be made elastic.

**Theorem 6.1.** *Let $N = 2^n$ and $\lambda$ be a security parameter, there exists an elastic polynomal commitment scheme for multiliear polynomials with at most n variables with the following properties:*

**Table 9:** Properties of elastic multilinear KZG scheme

| setup time | check time | commitment size | proof size |
|---|---|---|---|
| $O_\lambda(N)$ $\mathbb{G}$-ops | $O_\lambda(\log N)$ $\mathbb{G}$-ops | $O(1)$ | $O(\log N)$ |

where $\mathbb{G}$-ops denotes the number of group operations and $\mathbb{F}$-ops denotes the number of field operations. Moreover,

- in the **random access model**, the commitment algorithm takes $O_\lambda(N/\log N)$ $\mathbb{G}$-ops with space complexty $O(N)$, the opening algorithm takes $O(N)$ $\mathbb{F}$-ops and $O_\lambda(N)$ $\mathbb{G}$-ops with space complexity $O(N)$.
- in the **streaming model**, the commitment algorithm takes $O_\lambda(N)$ $\mathbb{G}$-ops with space complexity $O(1)$, the opening algorithm takes $O(N)$ $\mathbb{F}$-ops and $O_\lambda(N)$ $\mathbb{G}$-ops with space complexity $O(\log N)$. Both commitment and opening algorithms make one pass to the input streams.

### 6.1 Definition

**Definition 6.1.** A bilinear group is a tuple $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g, h, e)$, where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are groups of a prime order $p \in \mathbb{N}$, $g$ is the generator of $\mathbb{G}_1$, $h$ is the generator of $\mathbb{G}_2$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map satisfying the following properties:

- **bilinear:** for any $a, b \in \mathbb{Z}_p$, $e(g^a, h^b) = e(g, h)^{ab}$,
- **non-degenerate:** $e(g, h) \neq 1$.

Furthermore, the map $e$ is efficiently computable.

**Definition 6.2.** Let $\mathbb{G}$ be a group with order $p$, and let $\mathbf{g} = (g_1, \cdots, g_N) \in \mathbb{G}^N$ and $\mathbf{f} = (f_1, \cdots, f_N) \in \mathbb{Z}_p^N$. The **multi-scalar exponentiation** of $g$ and $f$ is given by $\prod_{i=1}^N g_i^{f_i}$. We denote the number of group operations required for performing an $O(N)$-sized multi-scalar exponentiation as $\mathrm{MSE}(N)$.

Let $\lambda$ be a security parameter, $\mathbb{G}$ be a group with order $p \approx 2^\lambda$. Performing one exponentiation operation in the group typically requires about $O(\lambda) = O_\lambda(1)$ group operations by the repeated squaring method. Pipenger proposed an algorithm for performing an $O(N)$-sized multi-scalar exponentiation with $\mathrm{MSE}(N) = O(N \cdot \lambda / \log(N \cdot \lambda)) = O_\lambda(N/\log N)$ group operations.

We present the syntax of the polynomial commitment scheme, noting that in our definition, the input to the algorithms is a vector $\mathbf{f}$ rather than a polynomial, as there is a one-to-one correspondence between a vector and its multilinear extension.

**Definition 6.3.** A multilinear polynomial commitment scheme over $\mathbb{F}$ is a tuple $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Check})$ with the following syntax:

- $\mathsf{PC}.\mathsf{Setup}\left(1^\lambda, n\right) \rightarrow (\mathsf{ck}, \mathsf{rk})$. On input a security parameter $\lambda$ (in unary) and a natural number $n$, where $n$ represents the maximum number of polynomial variables, PC.Setup samples a key pair $(\mathsf{ck}, \mathsf{rk})$.
- $\mathsf{PC}.\mathsf{Commit}(\mathsf{ck}, \mathbf{f}) \rightarrow C$. On input ck and a vector $\mathbf{f} \in \mathbb{F}^N$ where $N = 2^n$, PC.Commit outputs commitment $C$ to the vector $\mathbf{f}$ (actually, $C$ is a commitment to the multilinear polynomial $\widehat{\mathbf{f}}$).
- $\mathsf{PC}.\mathsf{Open}(\mathsf{ck}, \mathbf{f}, z) \rightarrow (\mu, \pi)$. On input ck, a vector $\mathbf{f} \in \mathbb{F}^N$ and a query point $z \in \mathbb{F}^n$, PC.Open outputs a evaluation $\mu = \widehat{\mathbf{f}}(z)$ and a proof $\pi$.
- $\mathsf{PC}.\mathsf{Check}(\mathsf{rk}, C, z, \mu, \pi) \rightarrow 0/1$. On input rk, the commitment $C$, query point $z \in \mathbb{F}^n$, alleged evaluation $\mu$, and an evaluation proof $\pi$, PC.Check outputs 1 if and only if $\pi$ attests that $\mu = \widehat{\mathbf{f}}(z)$.

## 6.2 Construction

**Theorem 6.2.** *Suppose $\mathbf{f} \in \mathbb{F}^N$ is a vector of length $N = 2^n$, $\widehat{\mathbf{f}}$ is the multilinear extension of $\mathbf{f}$, then for any $\mathbf{z} = (z_1, \cdots, z_n) \in \mathbb{F}^n$, $\widehat{\mathbf{f}}(\mathbf{z}) = \mu$ if and only if there exists a unique set of $n$ vectors $\mathbf{f}_1 \in \mathbb{F}^{N/2}, \cdots, \mathbf{f}_i \in \mathbb{F}^{N/2^i}, \cdots, \mathbf{f}_n \in \mathbb{F}$ such that*

$$\widehat{\mathbf{f}}(X_1, \cdots, X_n) - \mu = \sum_{i=1}^{n} (X_{n-i+1} - z_{n-i+1}) \cdot \widehat{\mathbf{f}}_i(X_1, \cdots, X_{n-i}), \tag{32}$$

*where $\widehat{\mathbf{f}}_i$ is the multilinear extension of $\mathbf{f}_i$, and $\widehat{\mathbf{f}}_n$ is a constant function with no variable.*

We prove Theorem 6.2 from the following lemma. Note that Lemma 6.3 not only implies Theorem 6.2 but also provides a construction method for vectors $\mathbf{f}_1, \cdots, \mathbf{f}_n$. This is crucial in our design of the streaming algorithm.

**Lemma 6.3.** *Suppose $N = 2^n$, $\mathbf{f} \in \mathbb{F}^N$ and $\mathbf{z} = (z_1, \cdots, z_n) \in \mathbb{F}^n$, for all $1 \le i \le n$, define vector $\mathbf{f}^{(i)} \in \mathbb{F}^{N/2^i}$ such that for all $\mathbf{v} \in \{0,1\}^{n-i}$ (see also Definition 4.2, we assume that $\mathbf{f}^{(0)} = \mathbf{f}$),*

$$\mathbf{f}^{(i)}(v_1, \cdots, v_{n-i}) = \widehat{\mathbf{f}}(v_1, \cdots, v_{n-i}, z_{n-i+1}, \cdots, z_n). \tag{33}$$

*Define vector $\mathbf{f}_i \in \mathbb{F}^{N/2^i}$ such that for all $\mathbf{v} \in \{0,1\}^{n-i}$,*

$$\mathbf{f}_i(v_1, \cdots, v_{n-i}) = \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 1) - \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 0). \tag{34}$$

*Then for all $1 \le i \le n$,*

$$\widehat{\mathbf{f}}^{(i-1)}(X_1, \cdots, X_{n-i+1}) = \widehat{\mathbf{f}}^{(i)}(X_1, \cdots, X_{n-i}) + (X_{n-i+1} - z_{n-i+1}) \cdot \widehat{\mathbf{f}}_i(X_1, \cdots, X_{n-i}). \tag{35}$$

**Proof.** Since $\widehat{\mathbf{f}}^{(i-1)}$ is multilinear, we have

$$\begin{aligned}&\widehat{\mathbf{f}}^{(i-1)}(X_1, \cdots, X_{n-i}, X_{n-i+1}) \\ &\quad = (1 - X_{n-i+1}) \cdot \widehat{\mathbf{f}}^{(i-1)}(X_1, \cdots, X_{n-i}, 0) + X_{n-i+1} \cdot \widehat{\mathbf{f}}^{(i-1)}(X_1, \cdots, X_{n-i}, 1),\end{aligned}$$

and

$$\widehat{\mathbf{f}}^{(i)}(X_1,\cdots,X_{n-i}) = \widehat{\mathbf{f}}^{(i-1)}(X_1,\cdots,X_{n-i},z_{n-i+1})$$
$$= (1-z_{n-i+1})\cdot\widehat{\mathbf{f}}^{(i-1)}(X_1,\cdots,X_{n-i},0) + z_{n-i+1}\cdot\widehat{\mathbf{f}}^{(i-1)}(X_1,\cdots,X_{n-i},1),$$

By subtracting these two polynomials, we get

$$\widehat{\mathbf{f}}^{(i-1)}(X_1,\cdots,X_{n-i},X_{n-i+1}) - \widehat{\mathbf{f}}^{(i)}(X_1,\cdots,X_{n-i})$$
$$= (X_{n-i+1}-z_{n-i+1})\cdot\left(\widehat{\mathbf{f}}^{(i-1)}(X_1,\cdots,X_{n-i},1) - \widehat{\mathbf{f}}^{(i-1)}(X_1,\cdots,X_{n-i},0)\right)$$
$$= (X_{n-i+1}-z_{n-i+1})\cdot\widehat{\mathbf{f}}_i(X_1,\cdots,X_{n-i}),$$

the second equality holds from the definition of $\mathbf{f}_i$ and from the fact that multilinear extension is unique. □

Since $\mathbf{f}^{(0)} = \mathbf{f}$ and $\mathbf{f}^{(n)} = \widehat{\mathbf{f}}(z_1,\cdots,z_n) = \mu$, Theorem 6.2 follows from Lemma 6.3 by a simple inductive argument.

The KZG polynomial scheme for multilinear polynomials consists of the following algorithms.

- PC.Setup $(1^\lambda, n) \to (\mathrm{ck},\mathrm{rk})$. On input a security parameter $\lambda$ (in unary), and a maximum number of polynomial variables $n$, PC.Setup samples a key pair $(\mathrm{ck},\mathrm{rk})$ as follows:
  - samples a bilinear group $(\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,p,g,h,e)$, where $p \approx 2^\lambda$.
  - samples $\boldsymbol{\tau} = (\tau_1,\cdots\tau_n) \in \mathbb{F}^n$,
  - for all $0 \le i < n$, computes
  $$\mathbf{T}_{\boldsymbol{\tau}}^{(i)} = (1-\tau_1,\tau_1)\otimes\cdots\otimes(1-\tau_{n-i},\tau_{n-i}) \in \mathbb{F}^{N/2^i},$$
  constructs
  $$\boldsymbol{\Sigma}_i = \left(g^{T^{(i)}_{\boldsymbol{\tau},0}}, g^{T^{(i)}_{\boldsymbol{\tau},1}},\cdots,g^{T^{(i)}_{\boldsymbol{\tau},N/2^i-1}}\right) \in \mathbb{G}_1^{N/2^i},$$
  where $N = 2^n$ and sets $\mathbf{T}_{\boldsymbol{\tau}}^{(n)} = (1)$, $\boldsymbol{\Sigma}_n = (g)$. Note that $T^{(i)}_{\boldsymbol{\tau},j}$ denotes $j$-th element of vetcor $\mathbf{T}_{\boldsymbol{\tau}}^{(i)}$, which can also be represented as $\mathbf{T}_{\boldsymbol{\tau}}^{(i)}(\mathbf{j})$ where $\mathbf{j} = \mathbf{toBinary}(j)$.
  - computes
  $$\boldsymbol{\Sigma}' = (h^{\tau_1},h^{\tau_2},\cdots,h^{\tau_n}) \in \mathbb{G}_2^n.$$
  - outputs $\mathrm{ck} = ((p,g,h,e),\boldsymbol{\Sigma}_0,\cdots,\boldsymbol{\Sigma}_{n-1},\boldsymbol{\Sigma}_n)$ and $\mathrm{rk} = ((p,g,h,e),\boldsymbol{\Sigma}')$.
- PC.Commit $(\mathrm{ck},\mathbf{f}) \to C$. On input ck and a vector $\mathbf{f} \in \mathbb{F}^N$ with length $N = 2^n$, PC.Commit outputs commitment $C$ to $\mathbf{f}$ as follows:
  - outputs $C = g^{\widehat{\mathbf{f}}(\boldsymbol{\tau})} = \prod_{\mathbf{j}\in\{0,1\}^n}\left(g^{\mathbf{T}_{\boldsymbol{\tau}}^{(0)}(\mathbf{j})}\right)^{\mathbf{f}(\mathbf{j})} \in \mathbb{G}_1$.
- PC.Open $(\mathrm{ck},\mathbf{f},\mathbf{z}) \to (\mu,\pi)$. On input ck, a vector $\mathbf{f} \in \mathbb{F}^N$ with length $N = 2^n$ and a query point $\mathbf{z} \in \mathbb{F}^n$, PC.Open outputs the evaluation $v$ and proof $\pi$ as follows:
  - computes $\mu = \widehat{\mathbf{f}}(\mathbf{z})$.
  - computes $n$ vectors $\mathbf{f}_1 \in \mathbb{F}^{N/2},\cdots,\mathbf{f}_i \in \mathbb{F}^{N/2^i},\cdots,\mathbf{f}_n \in \mathbb{F}$ as described in Theorem 6.2.
  - computes $y_i = g^{\widehat{\mathbf{f}}_i(\boldsymbol{\tau})} = \prod_{\boldsymbol{j}\in\{0,1\}^{n-i}}\left(g^{\mathbf{T}_{\boldsymbol{\tau}}^{(i)}(\mathbf{j})}\right)^{\mathbf{f}_i(\mathbf{j})}$ for all $0 < i \le n$.
  - outputs $(\mu,\pi)$, where $\pi = (y_1,\cdots,y_n) \in \mathbb{G}_1^n$.
- PC.Check $(\mathrm{rk},C,\mathbf{z},\mu,\pi) \to 0/1$. On input rk, the commitment $C$, query point $\mathbf{z} \in \mathbb{F}^n$, alleged evaluation $\mu$, and an evaluation proof $\pi$, PC.Check checks the proof as follows:
  - parses $\pi = (y_1,\cdots,y_n)$,

– checks that $e(C \cdot g^{-\mu}, h) = \prod_{i=1}^{n} e(y_i, h^{\tau_{n-i+1}} \cdot h^{-z_{n-i+1}})$.

**Lemma 6.4.** *In the KZG scheme for multilinear polynomials,* PC.Setup *takes $O_\lambda(N)$ group operations,* PC.Check *takes $O_\lambda(\log N)$ group operations and $O(\log N)$ pairings. Futhermore, the commit size is* 1 *and the proof contains $O(\log N)$ group elements.*

**Proof.** PC.Setup does not need to compute $\mathbf{T}_{\boldsymbol{\tau}}^{(i)}$ explicitly, instead, the PC.Setup can construct $\boldsymbol{\Sigma}_i$ from $\boldsymbol{\Sigma}_{i+1}$. When $i = n-1$, $\boldsymbol{\Sigma}_{n-1} = \left(g^{T_{\boldsymbol{\tau},0}^{(n-1)}}, g^{T_{\boldsymbol{\tau},1}^{(n-1)}}\right) = \left(g^{1-\tau_1}, g^{\tau_1}\right)$. Assume PC.Setup has

$$\boldsymbol{\Sigma}_{i+1} = \left(g^{T_{\boldsymbol{\tau},0}^{(i+1)}}, \cdots, g^{T_{\boldsymbol{\tau},N/2^{i+1}-1}^{(i+1)}}\right) \in \mathbb{G}_1^{N/2^{i+1}},$$

in hand, PC.Setup can construct $\boldsymbol{\Sigma}_i$ as follows:

$$\boldsymbol{\Sigma}_i = \left(g^{T_{\boldsymbol{\tau},0}^{(i+1)}\cdot(1-\tau_{n-i})}, g^{T_{\boldsymbol{\tau},0}^{(i+1)}\cdot\tau_{n-i}}, \cdots, g^{T_{\boldsymbol{\tau},N/2^{i+1}-1}^{(i+1)}\cdot(1-\tau_{n-i})}, g^{T_{\boldsymbol{\tau},N/2^{i+1}-1}^{(i+1)}\cdot\tau_{n-i}}\right) \in \mathbb{G}_1^{N/2^i}.$$

The correctness of the algorithm follows from the tensor structure of $\mathbf{T}_{\boldsymbol{\tau}}^{(i)}$, since

$$\mathbf{T}_{\boldsymbol{\tau}}^{(i)} = \mathbf{T}_{\boldsymbol{\tau}}^{(i+1)} \otimes (1-\tau_{n-i}, \tau_{n-i}).$$

Therefore, PC.Setup takes $2+4+\cdots+N+\log N = O(N)$ group exponentiations (thus $O_\lambda(N)$ group operations) to construct pk and ck.

In order to check $e(C \cdot g^{-\mu}, h) = \prod_{i=1}^{n} e(y_i, h^{\tau_{n-i+1}} \cdot h^{-z_{n-i+1}})$, PC.Check first computes $h^{-z_i}$ for all $1 \le i \le n$, which takes $O(n) = O(\log N)$ group exponentiations (thus $O_\lambda(\log N)$ group operations) in $\mathbb{G}_2$. Then computes $e(y_i, h^{\tau_{n-i+1}} \cdot h^{-z_{n-i+1}})$ for all $1 \le i \le n$ and $e(C \cdot g^{-\mu}, h)$, which takes $O(n) = O(\log N)$ pairings. The commit is only one group element $C$ in $\mathbb{G}_1$ and the proof contains $O(n) = O(\log N)$ group elements $(y_1, \cdots, y_n) \in \mathbb{G}^n$. □

**Lemma 6.5.** *In the random access model,* PC.Commit *takes* MSE$(N) = O_\lambda(N/\log N)$ *group operations,* PC.Open *takes $O(N)$ field operations and $\sum_{i=0}^{n-1}$* MSE$(2^i) = O_\lambda(N)$ *group operations.*

**Proof.** The complexity of PC.Commit is directly from the description of the scheme, which takes MSE$(N)$ group operations and $O(N)$ space complexity, since we assume ck and $\mathbf{f}$ are stored in the memory. For PC.Open, the prover uses Algorithm 4 for generating the evaluation $\mu$ and proof $\pi = (y_1, \cdots, y_n)$. The correctness of Algorithm 4 follows from Lemma 4.5 and Lemma 6.3. By the definition of multi-scalar exponentiation, $y_i = \prod_{\mathbf{j}\in\{0,1\}^{n-i}} \left(g^{\mathbf{T}_{\boldsymbol{\tau}}^{(i)}(\mathbf{j})}\right)^{\mathbf{f}_i(\mathbf{j})} = g^{\widehat{\mathbf{f}}_i(\boldsymbol{\tau})}$.

In the $i$-th round, PC.Open takes $O(2^{n-i})$ field operations to construct $\mathbf{f}_i$ and $\mathbf{f}^{(i)}$ and MSE$(2^{n-i})$ group operations to construct $y_i$. Totally, PC.Open takes $\sum_{i=1}^{n} O(2^{n-i}) = O(N)$ field operations and $\sum_{i=0}^{n-1}$ MSE$(2^i) = O_\lambda(N)$ group operations. □

**Lemma 6.6.** *In the streaming model,* PC.Commit *takes $O_\lambda(N)$ group operations with space complexity $O(1)$,* PC.Open *takes $O(N)$ field operations and $O_\lambda(N)$ group operations with space complexity $O(\log N)$. Both* PC.Commit *and* PC.Open *make one pass to the input streams.*

**Algorithm 4:** PC.Open in random access model

```
Input: (ck, f, z),
    • commit key $\mathrm{ck} = ((p, g, h, e), \mathbf{\Sigma}_0, \cdots, \mathbf{\Sigma}_n)$, all inputs are stored in the memory.
    • vector $\mathbf{f} \in \mathbb{F}^N$, which is stored in the memory.
    • query point $\mathbf{z} = (z_1, \cdots, z_n) \in \mathbb{F}^n$.
Output: Evaluation $\mu = \widehat{\mathbf{f}}(z)$, proof $\pi = (y_1, \cdots, y_n)$.
1  for $i \leftarrow 1$ to $n$ do
2      Initialize two vectors $\mathbf{f}_i, \mathbf{f}^{(i)} \in \mathbb{F}^{N/2^i}$;
3      for $v \leftarrow 0$ to $N/2^i - 1$ do
4          $\mathbf{f}_i(v_1, \cdots, v_{n-i}) \leftarrow \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 1) - \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 0)$;
           // $(v_1, \cdots, v_{n-i})$ is the binary representation of $v$.
5          $\mathbf{f}^{(i)}(v_1, \cdots, v_{n-i}) \leftarrow (1 - z_{n-i+1}) \cdot \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 0) + z_{n-i+1} \cdot \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 1)$;
6      end
7      $y_i \leftarrow$ multi-scalar exponentiation of $\mathbf{\Sigma}_i$ and $\mathbf{f}_i$;
8  end
9  $\mu \leftarrow \mathbf{f}^{(n)}$;
10 output $\mu, y_1, \cdots, y_n$;
```

**Proof.** In the streaming model, we assume that $\mathbf{\Sigma}_0, \cdots, \mathbf{\Sigma}_n$, $\mathbf{f}$ are provided to PC.Open as streams. The complexity of PC.Commit is directly from the description of the scheme, which takes $O(N)$ group exponentiations (thus $O_\lambda(N)$ group operations) and $O(1)$ space complexity.

The main challenge of PC.Open lies in the need to generate vectors $\mathbf{f}_1, \cdots, \mathbf{f}_n$ within a relatively small space overhead. The streaming algorithm for PC.Open is given in Algorithm 5. In lines 12-13, we pop two elements from the stack, suppose that $k_1 = k_2 = i - 1$ for some $i$, then both $elem_1$ and $elem_2$ are from the vector $\mathbf{f}^{(i-1)}$. In particular, $elem_1$ and $elem_2$ are two adjacent elements in $\mathbf{f}^{(i-1)}$. Assume that $elem_1 = \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 1)$ and $elem_2 = \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 0)$ (since $elem_1$ is popped first, $elem_1$ is in the "right" side of $elem_2$), then

$$\begin{aligned} elem &= (1 - z_{n-k_1}) \cdot elem_2 + z_{n-k_1} \cdot elem_1 \\ &= (1 - z_{n-i+1}) \cdot \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 0) + z_{n-i+1} \cdot \mathbf{f}^{(i-1)}(v_1, \cdots, v_{n-i}, 1) \\ &= \widehat{\mathbf{f}}^{(i-1)}(v_1, \cdots, v_{n-i}, z_{n-i+1}) \\ &= \mathbf{f}^{(i)}(v_1, \cdots, v_{n-i}) \end{aligned} \tag{36}$$

Therefore, we push $(elem, k_1 + 1) = (\mathbf{f}^{(i)}(v_1, \cdots, v_{n-i}), i)$ back to the stack. When $elem_1$ and $elem_2$ are popped, lines 15 - 21 have executed $v$ times under the condition $k_1 = k_2 = i - 1$, where $v$ is the decimal representation of $(v_1, \cdots, v_{n-i})$. Therefore, $\mathcal{S}(\mathbf{\Sigma}_{k_1+1}).next()$ will yield the $v + 1$-th element of $\mathbf{\Sigma}_{k_1+1}$, which is $g^{\mathbf{T}_\tau^{(i)}(v)}$, note that the index of $\mathbf{\Sigma}_{k_1+1}$ starts from 0. From the definition of vector $\mathbf{f}_i$, we know that $elem_1 - elem_2 = \mathbf{f}_i(v_1, \cdots, v_{n-i}) = \mathbf{f}_i(\mathbf{v})$. It follows that at the end of the algorithm, $y_i = \prod_{\mathbf{v} \in \{0,1\}^{n-i}} \left(g^{\mathbf{T}_\tau^{(i)}(\mathbf{v})}\right)^{\mathbf{f}_i(\mathbf{v})} = g^{\widehat{\mathbf{f}}_i(\tau)}$. When $k_1 = k_2 = n - 1$, $elem_1 = \mathbf{f}^{(n-1)}(1)$ and $elem_2 = \mathbf{f}^{(n-1)}(0)$, so $elem = (1 - z_1) \cdot \mathbf{f}^{(n-1)}(0) + z_1 \cdot \mathbf{f}^{(n-1)}(0) = \mathbf{f}^{(n)} = \mu$.

**Algorithm 5:** PC.Open in streaming model

**Input:** (ck, $\mathbf{f}$, $\mathbf{z}$),

- commit key ck $= ((p, g, h, e), \mathbf{\Sigma}_0, \cdots, \mathbf{\Sigma}_n)$, where $\mathbf{\Sigma}_0, \cdots, \mathbf{\Sigma}_n$ are all input in the form of streams $\mathcal{S}(\mathbf{\Sigma}_0), \cdots, \mathcal{S}(\mathbf{\Sigma}_n)$,
- vector $\mathbf{f} \in \mathbb{F}^N$, which is input in the form of stream $\mathcal{S}(\mathbf{f})$,
- query point $\mathbf{z} = (z_1, \cdots, z_n) \in \mathbb{F}^n$.

**Output:** evaluation $\mu = \widehat{\mathbf{f}}(\mathbf{z})$, proof $\pi = (y_1, \cdots, y_n)$.

```
1  Init a stack $s$ ;   // The elements of the stack $s$ are pairs $(elem, k)$, where $elem \in \mathbb{F}$ and $0 \le k \le n$, $elem$ is in the vector $\mathbf{f}^{(k)}$.
2  for $i \leftarrow 0$ to $n$ do
3      $\mathcal{S}(\mathbf{\Sigma}_i).init()$;
4      $y_i \leftarrow 1$;
5  end
6  $\mathcal{S}(\mathbf{f}).init()$;
7  while (true) do
8      if $s.size() < 2$ then
9          $elem \leftarrow \mathcal{S}(\mathbf{f}).next()$;
10         $s.push(elem, 0)$ ;   // If the stack has at most 1 element, push a new element from $\mathcal{S}(\mathbf{f})$ into stack.
11     else
12         $(elem_1, k_1) \leftarrow s.pop()$;
13         $(elem_2, k_2) \leftarrow s.pop()$;
14         if $k_1 = k_2$ then
15             $elem \leftarrow (1 - z_{n-k_1}) \cdot elem_2 + z_{n-k_1} \cdot elem_1$;
16             $s.push(elem, k_1 + 1)$ ;
17             $w \leftarrow \mathcal{S}(\mathbf{\Sigma}_{k_1+1}).next()$;
18             $y_{k_1+1} \leftarrow y_{k_1+1} \cdot w^{elem_1 - elem_2}$;
19             if $k_1 == n - 1$ then
20                 output $\mu = elem, y_1, \cdots, y_n$ and stop ;   // Output proof and stop running
21             end
22         else
23             $s.push(elem_2, k_2)$;
24             $s.push(elem_1, k_1)$;
25             $elem \leftarrow \mathcal{S}(\mathbf{f}).next()$;
26             $s.push(elem, 0)$;
27         end
28     end
29 end
```

The space cost of Algorithm 5 is primarily from the stack and the storage of $y_1, \cdots, y_n$. Each vector $(\mathbf{f}^{(0)}, \cdots, \mathbf{f}^{(n)})$ has at most two elements in the stack, otherwise the algorithm will pop them and merge them into a new element (line 15). Therefore the space complexity of the algorithm is $O(n) = O(\log N)$. Algorithm 5 makes one pass to each input streams $\mathcal{S}(\mathbf{\Sigma}_0), \cdots, \mathcal{S}(\mathbf{\Sigma}_n), \mathcal{S}(\mathbf{f})$. PC.Open takes $O(N)$ field operations for constructing $\mathbf{f}^{(1)}, \cdots, \mathbf{f}^{(n)}$, since there are $N/2^i = 2^{n-i}$ elements in $\mathbf{f}^{(i)}$ and every element takes constant time to construct (line 15). Line 18 will be executed $N/2^i$ times since $y_i = \prod_{\mathbf{v}\in\{0,1\}^{n-i}} \left(g^{\mathbf{T}_{\boldsymbol{\tau}}^{(i)}(\mathbf{v})}\right)^{\mathbf{f}_i(\mathbf{v})}$, so PC.Open takes $O(N/2^i)$ group exponentiations for computing $y_i$. Totally, PC.Open takes $\sum_{i=1}^{n} O(N/2^i) = O(N)$ group exponentiations (thus $O_\lambda(N)$ group operations) for generating $y_1, \cdots, y_n$. □

## 7 Implementation and Evaluation

We implemented Epistle in Rust by leveraging open-source libraries such as arkworks[1] and hyperplonk[2]. Our implementation includes the elastic PIOP from Section 4 (which comprises the SumCheck, ZeroCheck, Cyclic Shift Left check, Product check, and Permutation check subprotocols) and the elastic multilinear KZG polynomial commitment scheme from Section 6.

### 7.1 Stream infrastructure

Similar to Gemini, we also utilized a wrapper over iter::Iterator to express stream, which can be restarted and iterated over multiple times. The item generated by the stream implemented the borrow trait so that we can avoid copying elements through rust's borrow abstraction.

### 7.2 Optimaztions

**Elastic prover.** Epistle also supports switching from the space-efficient implementation to the time-efficient implementation with specified memory threshold like Gemini. In SumCheck PIOP, if the elastic prover has enough memory to proving current round, then it can switch the space-efficient prover state to time-efficient prover state. This allows the elastic prover to speed up last few rounds of proving by time-efficient function and Since the prover's messages are the same in both modes, the final proof is identical.

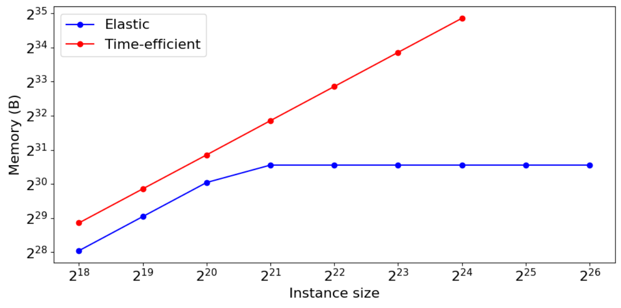
**Batch elastic multilinear KZG.** HyperPlonk proposed a batch opening protocol to batch evaluation proofs for a set of evaluation points over different multivariate polynomials. We implement these to our elastic multilinear KZG polynomial commitment scheme for batch opening, resulting in reduced opening and verification time.

[1]https://github.com/arkworks-rs

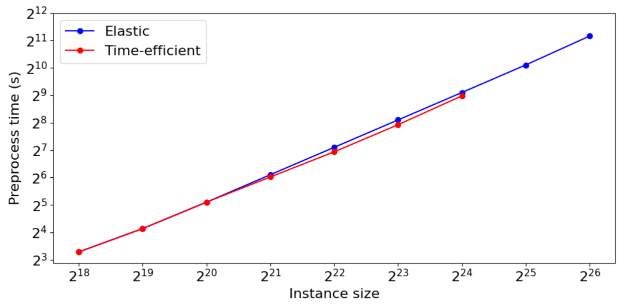
[2]https://github.com/EspressoSystems/hyperplonk

## 7.3 Evaluation

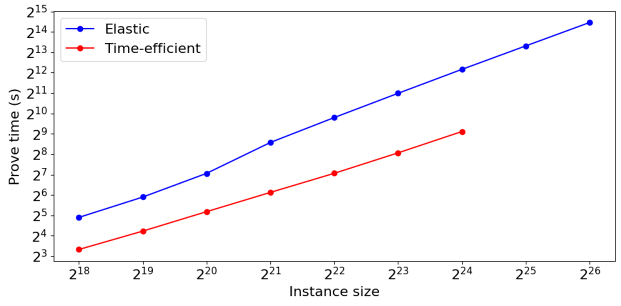
We run benchmarks for Epistle over a 2023 Apple MacBook Pro with M3 Max chip, 14 cores and 36GB RAM. We use vanilla plonk gate for mock circuit with BLS12-381 elliptic curve. Benchmarking was performed with instance sizes ranging from $2^{18}$ to $2^{26}$.



**(a)** Memory vs. Instance size



**(b)** Preprocess time vs. Instance size



**(c)** Prove time vs. Instance size

**Figure 1:** (a) memory usage, (b) preprocess time and (c) prove time for the elastic prover(blue) and time-efficient prover(red) for different instance sizes.

**Proving space.** The memory usage is obtained by DHAT, a memory profiler for Rust program. As shown in Figure 1: (a), the elastic prover memory usage remains constant at approximately 1.5GB as instance size increases, while the time-efficient prover will trigger out of memory(OOM) at instance size $2^{25}$ in our 36GB RAM. Two main parameters affect memory usage:

- **Multi-scalar multiplication (MSM) buffer.** MSM algorithms (e.g., Pippenger's algorithm) improve time efficiency by allocating memory buffer for data. In our implementation, we use one MSM buffer for commit phase and $\log N$ MSM buffers for open phase. To avoid impacting peak memory usage, we set the commit phase buffer size to $2^{20}$ and the open phase buffer size to $2^{17}$.
- **Sumcheck round threshold.** In our implementation, we set the threshold to 20, meaning that the time-efficient prover is adopted in the last 20 rounds. According to our experimental results, for instance sizes greater than $2^{20}$, the peak memory usage was caused by the sumcheck round threshold of 20.

HyperPlonk provides benchmarks for sizes up to $2^{20}$ and our benchmark for time-efficient snark stop at $2^{24}$ due to out of memory crashes. Pianist benchmarks instance sizes up to $2^{25}$ in the Plonk constraint system, consuming over 100GB of memory on a single machine and 5GB of memory per machine in a 32-machine distributed setup. We benchmark Epistle and increase instance size to $2^{26}$ while consuming only 1.5GB of memory, but the instance upper limit in our benchmarks is arbitrary as long as it can generate input streams for proving.

**Preprocessing time.** The preprocessing phase before proving involves building oracles and committing to them. As shown in Figure 1: (b), we can observe that the results in both modes are linear and very close to each other.

**Proving time.** The elastic prover can switch to time-efficient prover, and if the instance size is less than $2^{20}$, elastic prover will switch to time-efficient mode in the initial phase. Note that the transition from space-efficient to time-efficient will take some time. In Figure 1: (c), We show the proving time in elastic and time-efficient for different instance sizes respectively and we observe that the proving time is almost linear in both modes. As the instance size increases from $2^{18}$ to $2^{24}$, the time difference between the two modes is about 2-8x.

**Proof size and verification time.** For instance sizes ranging from $2^{18}$ to $2^{26}$, the proof size is about 14 - 19 KB and the verification time is about 10 - 15 ms.

## Acknowledgement

The authors would like to thank Yang Xu and Liguan Wang for helpful discussions on this work. This work was supported by National Natural Science Foundation of China (Grant No. 62272107), the Innovation Action Plan of Shanghai Science and Technology (No. 21511102200), and the Key R & D Program of Guangdong Province (No. 2020B0101090001).

## References

[1] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC)*, pages 723–732, 1992.

[2] Silvio Micali. CS proofs. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 436–453, 1994.

[3] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the 43th Annual ACM Symposium on Theory of Computing (STOC)*, pages 99–108, 2011.

[4] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS)*, pages 326–349, 2012.

[5] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive, Report 2019/953*, 2019.

[6] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Proceedings of the 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 499–530, 2023.

[7] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. *Cryptology ePrint Archive, Report 2023/1271*, 2023.

[8] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orru. Gemini: Elastic snarks for diverse environments. In *Proceedings of the 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 427–457, 2022.

[9] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Proceedings of the 14th Theory of Cryptography Conference (TCC)*, pages 31–60, 2016.

[10] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 738–768, 2020.

[11] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 677–706, 2020.

[12] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 177–194, 2010.

[13] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings of the 6th Annual International Cryptology Conference (CRYPTO)*, pages 186–194, 1986.

[14] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In *Proceedings of the 11st Theory of Cryptography Conference (TCC)*, pages 222–242, 2013.

[15] Scroll. https://scroll.io/.

[16] Ariel Gabizon and Zachary J Williamson. plookup: A simplified polynomial protocol for lookup tables. *Cryptology ePrint Archive, Report 2020/315*, 2020.

[17] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Proceedings of the 42nd Annual International Cryptology Conference (CRYPTO)*, pages 299–328, 2022.

[18] Daniel A Spielman. Linear-time encodable and decodable error-correcting codes. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 388–397, 1995.

[19] Louay MJ Bazzi and Sanjoy K Mitter. Endcoding complexity versus minimum distance. *IEEE Transactions on Information Theory*, 51(6):2103–2112, 2005.

[20] Anna Gál, Kristoffer Arnsfelt Hansen, Michal Kouckỳ, Pavel Pudlák, and Emanuele Viola. Tight bounds on computing error-correcting codes by bounded-depth circuits with arbitrary gates. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 479–494, 2012.

[21] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. *Cryptology ePrint Archive, Report 2023/552*, 2023.

[22] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive, Report 2018/046*, 2018.

[23] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Proceedings of the 40th Annual International Cryptology Conference (CRYPTO)*, pages 704–737, 2020.

[24] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70–122, 1998.

[25] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[26] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, pages 238–252, 2013.

[27] Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *Proceedings of the 23th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 336–365, 2017.

[28] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In *Proceedings of the 18th Theory of Cryptography Conference (TCC)*, pages 19–46, 2020.

[29] Jonathan Bootle, Alessandro Chiesa, and Siqi Liu. Zero-knowledge IOPs with linear-time prover and polylogarithmic-time verifier. In *Proceedings of the 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 275–304, 2022.

[30] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S Wahby. Brakedown: Linear-time and post-quantum SNARKs for R1CS. In *Proceedings of the 43rd Annual International Cryptology Conference (CRYPTO)*, 2023.

[31] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 111–120, 2013.

[32] Justin Holmgren and Ron Rothblum. Delegating computations with (almost) minimal time and space overhead. In *Proceedings of the 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 124–135, 2018.

[33] Alexander R Block, Justin Holmgren, Alon Rosen, Ron D Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In *Proceedings of the 18nd Theory of Cryptography Conference (TCC)*, pages 168–197, 2020.

[34] Alexander R Block, Justin Holmgren, Alon Rosen, Ron D Rothblum, and Pratik Soni. Time-and space-efficient arguments from groups of unknown order. In *Proceedings of the 41st Annual International Cryptology Conference (CRYPTO)*, pages 123–152, 2021.

[35] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the 39th IEEE symposium on security and privacy (S&P)*, pages 315–334, 2018.

[36] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, 1992.

[37] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS)*, pages 90–112, 2012.

[38] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the 33th Annual International Cryptology Conference (CRYPTO)*, pages 71–89, 2013.

[39] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zksnarks. *Cryptology ePrint Archive, Report 2020/1275*, 2020.

[40] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[41] László Babai. Trading group theory for randomness. In *Proceedings of the 17th annual ACM Symposium on Theory of Computing (STOC)*, pages 421–429, 1985.