SCRIBE: Low-memory SNARKs via Read-Write Streaming

Anubhav Baweja
abaweja@upenn.edu
UPenn

Pratyush Mishra
prat@upenn.edu
UPenn

Tushar Mopuri
tmopuri@upenn.edu
UPenn

Karan Newatia
knewatia@upenn.edu
UPenn

Steve Wang
qwang97@upenn.edu
UPenn

December 5, 2024

Abstract

Succinct non-interactive arguments of knowledge (SNARKs) enable a prover to produce a short and efficiently verifiable proof of the validity of an arbitrary NP statement. Recent constructions of efficient SNARKs have led to interest in using them for a wide range of applications, but unfortunately, deployment of SNARKs in these applications faces a key bottleneck: SNARK provers require a prohibitive amount of time and memory to generate proofs for even moderately large statements. While there has been progress in reducing prover time, prover memory remains an issue.

In this work, we describe SCRIBE, a new *low-memory* SNARK that can efficiently prove large statements even on cheap consumer devices such as smartphones by leveraging a plentiful, but heretofore unutilized, resource: disk storage. In more detail, instead of storing its (large) intermediate state in RAM, SCRIBE's prover instead stores it on disk. To ensure that accesses to state are efficient, we design SCRIBE's prover in a *read-write streaming model* of computation that allows the prover to read and modify its state only in a streaming manner.

We implement and evaluate SCRIBE's prover, and show that, on commodity hardware, it can easily scale to circuits of size 2^{28} gates while using only 2 GB of memory and incurring only minimal proving latency overhead (10-35%) compared to a state-of-the-art memory-intensive baseline (HyperPlonk [EUROCRYPT 2023]) that requires much more memory. Our implementation minimizes overhead by leveraging the streaming access pattern to enable several systems optimizations that together mask I/O costs.

Contents

1 Introduction

SNARKs enable a prover to convince a verifier of the validity of correct program execution via a *succinct* proof that can be checked much more quickly than running the program itself. There has been much recent interest in the construction of efficient SNARKs for a wide range of applications, including blockchain rollup systems [Whi18] and cryptocurrency bridges [Xie+22]. Many of these applications require proving the correctness of large computations. For instance, both the rollup and bridge applications require proving satisfiability of circuits with billions of gates. Unfortunately, existing SNARKs incur large time and space overheads when proving large computations, requiring the use of powerful machines to generate proofs. For instance, recent industry benchmarks rely on server-class machines with powerful GPUs[1] and hundreds of gigabytes of RAM[2] to prove such statements.

Much effort [BCGJM18; Set20; CBBZ23; GLSTW23; HLP24; Pol; AST24] has been devoted to reducing the time overhead of SNARKs. While these efforts have greatly reduced prover latency, they do not address the high memory overheads. This overhead occurs because the size of the prover's internal state scales linearly with the size of the computation, as opposed to scaling with the (potentially much smaller) space complexity of the computation itself. For example, the HyperPlonk SNARK [CBBZ23] requires over $16\,\mathrm{GB}$ of RAM just to prove that $10\,\mathrm{kB}$ of data was hashed correctly with SHA256. As a result, these SNARKs cannot be used in systems with limited memory.

This has motivated recent efforts to reduce these memory requirements. These efforts all proceed by reducing the size of the prover's internal state via disparate techniques such as complexity-preserving SNARKs [BC12; BCCT13; BBHV22], SNARKs with streaming provers [BHRRS20; BHRRS21; BCHO22; ZCLKZ24], and recursive composition [BCTV14; BCMS20; BCLMS21; BDFG21; KST22]. However, as we explain in detail in Section 3, these approaches achieve lower memory usage only by sacrificing prover latency (both asymptotically and concretely). Moreover, some approaches even seem to require an inherent space-time tradeoff [BBHV22; CM24].

In sum, we do not have concretely efficient SNARKs that can prove large computations on commodity devices quickly without requiring a prohibitive amount of memory.

1.1 Our results

In this work, we tackle the foregoing problem via a new approach: instead of trying to reduce the size of the prover's internal state, we propose to instead change where it is stored and how it accessed by the prover. We formalize our approach via a new way to model low-memory algorithms, and construct in this model a new SNARK, SCRIBE, that effectively scales to large computations even on commodity devices. We detail our contributions below.

Read-write streaming. We introduce a new algorithm design framework which we call the *read-write streaming model*. Algorithms in this model have access to a small amount of random-access memory (e.g., RAM), and a large amount of external storage (e.g., disk) that stores *streams* containing the algorithm's state. These streams can be read and modified only sequentially from beginning to end.

Our model is motivated by the observation that while RAM is expensive and limited, disk storage is plentiful and cheap, and so it is natural to store an algorithm's (possibly large) internal state on disk. However, because random disk accesses are much costlier than RAM accesses, a naive attempt to port an algorithm to our model could incur significant latency overheads due to I/O costs. To avoid this, our model restricts the

[1] https://www.risczero.com/blog/beating-moores-law-with-zkvm-1-0.
[2] https://blog.succinct.xyz/sp1-is-live/.

algorithm's disk accesses to follow a predictable and data-independent *streaming* access pattern. This in turn enables numerous systems optimizations such as prefetching, pipelining, and caching that mask I/O costs.

We formalize read-write streams and algorithms that use them, provide efficiency measures for such algorithms, and describe how to efficiently compose them to minimize time and space overheads. We use this model to construct a new 'read-write streaming SNARK' that we describe next.

SCRIBE: a linear-time read-write streaming SNARK. We construct SCRIBE, a SNARK for arithmetic circuit satisfiability with a read-write streaming prover. To prove satisfiability of a circuit of size N, SCRIBE's prover requires: (i) $O(N)$ cryptographic (group) operations and $O(N)$ field operations, (ii) $O(\log N)$ random-access memory, and (iii) $O(N)$ external storage. SCRIBE is based on the HyperPlonk SNARK [CBBZ23], and preserves the latter's prover and verifier time complexity and succinct proof size, while reducing the prover's random-access memory from $O(N)$ to $O(\log N)$.

We obtain SCRIBE by adapting the popular "Polynomial IOP + Polynomial Commitment $\rightarrow$ SNARK" paradigm [CHMMVW20; BFS20] to the read-write streaming model. To do so, we first construct read-write streaming versions of polynomial IOPs (PIOPs) that are commonly used as building blocks in the literature, and show how to combine these to obtain a read-write streaming prover for HyperPlonk's PIOP. We also construct read-write streaming provers for a variety of popular polynomial commitment schemes [PST13; BGH19; WTSTW18; BMMTV21; Lee21]. All our PIOP and PC constructions preserve the time complexity of their non-streaming counterparts, while reducing their random-access space to logarithmic.

Implementation. We implement SCRIBE as a Rust library based on the `arkworks` framework [con22]. Our implementation uses CPU RAM for random-access, and relies on disk storage for externally stored streams. To efficiently work with these streams, we develop new abstractions over file I/O that enable features such as prefetching, batch operations, and parallelization. We provide details about our implementation and optimizations in Section 2.9.

Evaluation. We perform a thorough evaluation of SCRIBE's performance, and show that it can scale to prove circuits of size 2^{28} gates in just $1.5\,\mathrm{h}$ on a server-class machine. Our evaluation demonstrates that assuming reasonable hardware, read-write streaming imposes minimal overheads (10-25%) over the memory-intensive baseline of HyperPlonk. Furthermore, compared to a prior state-of-the-art low-memory SNARK [BCHO22], SCRIBE improves proving latency by almost an order of magnitude.

We also evaluate SCRIBE's performance on a commodity smartphone, and show that it can scale to prove computations $32\times$ larger than the memory-intensive baseline. We believe that this is the largest circuit that has been proven on a smartphone-class device to date.

2 Techniques

We describe the main ideas underlying SCRIBE.We begin by describing notation that we use in the following sections in Section 2.1. Then, in Section 2.2, we recall the popular 'PIOP + PC $\rightarrow$ SNARK' methodology [CHMMVW20; BFS20] of constructing SNARKs, and also recall how the HyperPlonk SNARK [CBBZ23], a state-of-the-art memory-intensive SNARK that achieves cryptographic linear time, is constructed using this methodology. HyperPlonk will serve as the starting point for our construction of SCRIBE.

Next, in Section 2.3, we introduce the notion of read-write (RW) streaming algorithms. We show in Section 2.4 how to construct RW streaming prover algorithms for 'PIOP + PC' SNARKs from RW streaming algorithms for the underlying components. We then proceed to construct the latter in Sections 2.5 and 2.6 (PIOPs) and Section 2.8 (PC schemes). Finally, in Section 2.9, we describe our implementation of SCRIBE, and evaluate its performance in Section 2.10.

The RW streaming algorithms we construct will often be derived in a simple way from their non-streaming counterparts. We view this simplicity as a strength of our approach, as it shows that our model is expressive enough to capture existing state-of-the-art algorithms, while also showing that these algorithms can be implemented with little random-access space with minimal time overhead.

2.1 Notation

We introduce some notation that we will use in the rest of this paper.

Vector and set notation. We use $[a_1, \ldots, a_n]$ to denote *ordered* sets, and denote the set $[1, 2, \ldots, n]$ by $[n]$. Let S be a finite set. An N-dimensional vector of elements is denoted by $\boldsymbol{x} \in S^N$, and x_i denotes the i-th element of the vector. Vectors are indexed as $\boldsymbol{x} = [x_i]_{i=1}^N = [x_1, x_2, \ldots, x_N]$.[3] We denote matrices by $\mathbf{M} \in S^{N \times N}$, where $\boldsymbol{M}_i$ represents the i-th row of the matrix. Given a vector $\boldsymbol{x} = [x_i]_{i=1}^N$, we use $\boldsymbol{x}_{[i:j]}$ to denote the vector $[x_k]_{k=i}^j$ and $y \cdot \boldsymbol{x}$ to denote the vector $[y \cdot x_i]_{i=1}^N$ for $y \in S$. $x \xleftarrow{\$} S$ denotes sampling x uniformly at random from a finite set S.

Vector partitions. For a vector $\boldsymbol{x} \in \mathbb{F}^N$, $\boldsymbol{x}_E = [x_{2i}]_{i=0}^{N/2-1}$ and $\boldsymbol{x}_O = [x_{2i+1}]_{i=0}^{N/2-1}$ are vectors containing the even-indexed and odd-indexed elements of $\boldsymbol{x}$ respectively, while $\boldsymbol{x}_L = [x_i]_{i=0}^{N/2-1}$ and $\boldsymbol{x}_R = [x_i]_{i=N/2}^{N-1}$ are vectors containing the left and right halves of $\boldsymbol{x}$ respectively.[4]

Binary representation. Given integers i and n, $\mathsf{bin}_n(i)$ represents the n-bit binary representation of i in the most-significant-bit order, and $\mathsf{bin}_n(i, j)$ represents the j-th bit of $\mathsf{bin}_n(i)$. We omit n when it is clear from context. We also define the inverse operation $\mathsf{int}(\boldsymbol{x})$ which maps $\boldsymbol{x} = \mathsf{bin}_n(v) \in \{0,1\}^n$ to v.

Multilinear polynomials. A polynomial is *multilinear* if the degree of each variable in every monomial is at most 1. A multilinear polynomial f is uniquely defined by its evaluations $\boldsymbol{f} := [f(\mathsf{bin}_n(i))]_{i=0}^{2^n-1}$ over the n-dimensional boolean hypercube $\{0,1\}^n$. We sometimes denote a polynomial $f(X_1, \ldots, X_n)$ by $f(\boldsymbol{X})$ when n is clear from context.

Multilinear extension. The *multilinear extension* of a vector $\boldsymbol{v} \in \mathbb{F}^{2^n}$ is the unique multilinear polynomial $\hat{v}(\boldsymbol{X}) \in \mathbb{F}[X_1, \ldots, X_n]$ such that $\hat{v}(\boldsymbol{i}) = v_{\mathsf{int}(\boldsymbol{i})}$ for all $\boldsymbol{i} \in \{0,1\}^n$.

Lagrange basis polynomial. The *Lagrange basis polynomial* $\mathrm{eq}_n(\boldsymbol{X}, \boldsymbol{Y}) := \prod_{i=1}^n (X_i Y_i + (1-X_i)(1-Y_i))$ checks that $\boldsymbol{X} = \boldsymbol{Y}$ for any $\boldsymbol{X}, \boldsymbol{Y} \in \{0,1\}^n$. We omit n when it is clear from context.

[3] We make an exception when indexing vectors corresponding to coefficients of multilinear polynomials; these are indexed as $\boldsymbol{x} = [x_i]_{i=0}^{N-1}$.

[4] These definitions assume that N is even and the vector is indexed from 0, which is true whenever they are used.

Oracle access. For an n-variate polynomial $p \in \mathbb{F}[\boldsymbol{X}]$, $[\![p]\!]$ denotes an *oracle* for p that can be queried at any point $\boldsymbol{x} \in \mathbb{F}^n$ to receive the evaluation $p(\boldsymbol{x}) \in \mathbb{F}$.

Multisets. We use the double braces notation $\{\!\{\cdot\}\!\}$ to represent multisets.

Algebraic notation. We use additive notation for groups. We rely on bilinear groups sampled by a randomized algorithm $\mathsf{SampleGrp}$ that outputs a tuple $(\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G, H)$ where $\mathbb{F}$ is a field of prime order q, $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are groups of order q, $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear map, G generates $\mathbb{G}_1$, and H generates $\mathbb{G}_2$.

Inner products. We use three types of inner products: (a) the scalar product $\langle \boldsymbol{x}, \boldsymbol{y} \rangle_{\mathbb{F}} := \sum_{i=1}^{N} x_i y_i$, (b) the group inner product $\langle \boldsymbol{x}, \boldsymbol{Y} \rangle_{\mathbb{G}} := \sum_{i=1}^{N} x_i \cdot Y_i$, and (c) the pairing inner product $\langle \boldsymbol{X}, \boldsymbol{Y} \rangle_e := \sum_{i=1}^{N} e(X_i, Y_i)$. We omit subscripts when it is clear from context which inner product is being used.

2.2 Starting point: HyperPlonk

To construct an efficient read-write streaming SNARK, our starting point will be the memory-*inefficient* SNARK of HyperPlonk [CBBZ23]. We choose this starting point because:

- asymptotically, HyperPlonk achieves *cryptographic* linear time[5]: it requires just $O(N)$ field operations and $O(N)$ group operations to prove satisfaction of an arithmetic circuit of size N.
- concretely, HyperPlonk achieves better prover times than almost all prior SNARKs, and additionally supports attractive features such as custom gates [GW19].

Background: SNARKs from Polynomial IOPs and PC schemes. The SNARK of HyperPlonk (as well as SCRIBE) is obtained via the methodology introduced by Chiesa et al. [CHMMVW20] and Bünz et al. [BFS20], which constructs SNARKs from two ingredients: Polynomial Interactive Oracle Proofs (PIOPs) and Polynomial Commitment (PC) schemes.

- *PIOPs* are interactive proofs where the prover's messages are *polynomial oracles*. The verifier does not read these messages in their entirety, but instead queries these polynomials at evaluation points of its choice. The verifier also often has oracle access to polynomial representations of the NP statement being proved; such PIOPs are called holographic PIOPs [CHMMVW20]. For example, a PIOP for circuit satisfiability provides an oracle representation of the circuit to the verifier.
- *PC schemes* are commitment schemes which enable the prover to commit to a polynomial p, and then later 'open' the commitment to prove the claim "$p(z) = v$", where z is an evaluation point chosen by the verifier and v is the prover's claimed evaluation of p at z.

The framework of [CHMMVW20; BFS20] composes these ingredients to construct succinct arguments as follows. First, if the PIOP verifier is supposed to have oracle access to a polynomial encoding of the NP statement, then the argument's preprocessing phase commits to this polynomial using the PC scheme, and provides this commitment to the argument verifier. The argument prover $\mathcal{P}$ and verifier $\mathcal{V}$ then engage in an interaction: in each round, $\mathcal{P}$ invokes the PIOP prover P for that round, commits to the polynomials produced by P via the PC scheme, and sends these commitments to $\mathcal{V}$. $\mathcal{V}$ then invokes the PIOP verifier V to compute its message for that round, and sends this to $\mathcal{P}$. At the end of the interaction, when V wishes to query its polynomial oracles at certain evaluation points, $\mathcal{V}$ sends these points to $\mathcal{P}$, which replies with the corresponding evaluation values and evaluation proofs for these using the PC scheme. A SNARK can then be constructed by invoking the Fiat–Shamir transform [FS86] on this interactive argument.

[5] In this paper, we say that an algorithm requires 'cryptographic X time' if it performs $O(X)$ group operations, Some prior work omits the distinction between cryptographic and non-cryptographic operations, but this is inaccurate [GLSTW23] because some group operations, in particular multi-scalar multiplications, are asymptotically super-linear even with the best known algorithms [Pip80].

HyperPlonk [CBBZ23] constructs a SNARK via this methodology by constructing a PIOP for circuit satisfiability. The PIOP relies on *multilinear* polynomial oracles, and so the corresponding PC scheme used in HyperPlonk is one that supports committing to these. We briefly recall HyperPlonk's constructions of these ingredients, starting with an overview of their circuit representation.

HyperPlonk's circuit representation. Consider an arithmetic circuit $C : \mathbb{F}^n \to \mathbb{F}$ with m gates, each of which is a fan-in 2 addition or multiplication gate.[6] HyperPlonk represents C as three vectors $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o} \in \mathbb{F}^m$, where ℓ_i, r_i, o_i denote the left input, right input, and output of the i-th gate in the circuit, respectively. The circuit is satisfied if and only if the following constraints are satisfied:

1. *Gate satisfaction constraints*: enforce that the gate operations are applied correctly. If the i-th gate is an addition gate, then $o_i = \ell_i + r_i$; if it is a multiplication gate, then $o_i = \ell_i \cdot r_i$.
2. *Wiring constraints*: enforce that wires between gates are connected correctly. For example, if the output of gate j is the left input of gate i, then there is a wiring constraint that enforces that $\ell_i = o_j$.

To encode this circuit representation in polynomial form, HyperPlonk associates with each circuit the following vectors: (a) a *selector* vector $\boldsymbol{s} \in \mathbb{F}^m$ such that s_i is 0 if the i-th gate is an addition gate, and is 1 if it is a multiplication gate, and (b) *permutation* vectors $\boldsymbol{\pi}, \boldsymbol{\sigma} \in \mathbb{F}^m$ that encode the wiring constraints as follows: if the left input of gate i is the output of gate j, then $\pi_i = j$, and similarly, if the right input of gate i is the output of gate j then $\sigma_i = j$.[7]

HyperPlonk's PIOP. In a preprocessing phase, the circuit C is arithmetized by encoding the vectors $\boldsymbol{s}$, $\boldsymbol{\pi}$, $\boldsymbol{\sigma}$ as multilinear polynomials $\hat{s}, \hat{\pi}, \hat{\sigma}$.[8] The PIOP prover receives these polynomials as explicit input, while the PIOP verifier is given oracle access to them.

During the interactive phase, the PIOP prover receives as additional input the witness vectors $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$, and proceeds as follows. The PIOP prover computes the multilinear extensions $\hat{\ell}$, $\hat{r}$, $\hat{o}$ of the witness vectors, and sends oracles for these to the PIOP verifier. The PIOP prover and verifier than engage in two subPIOPs corresponding to each of the checks above.

1. *Gate satisfaction*: The gate constraints are satisfied if and only if for all $i \in \{0, \ldots, m-1\}$, it holds that $s_i \cdot (o_i - \ell_i - r_i) + (1 - s_i) \cdot (o_i - \ell_i \cdot r_i) = 0$. It leverages this idea by encoding these checks as the polynomial $p = \hat{s} \cdot (\hat{o} - \hat{\ell} - \hat{r}) + (1 - \hat{s}) \cdot (\hat{o} - \hat{\ell} \cdot \hat{r})$, and enforcing that $p(\boldsymbol{i}) = 0$ for all $\boldsymbol{i} \in \{0,1\}^{\log m}$. This check is done via a *zerocheck* PIOP that enforces that a given polynomial evaluates to zero at all points in the boolean hypercube.
2. *Wiring*: The wiring constraints are satisfied if and only if for all $i \in [m]$, the left input ℓ_i of the i-th gate is the output o_{π_i} of the π_i-th gate, and the right input r_i of the i-th gate is the output o_{σ_i} of the σ_i-th gate. That is, $\boldsymbol{\ell}$ is a permutation of $\boldsymbol{o}$ with respect to $\boldsymbol{\pi}$, and similarly for $\boldsymbol{r}$ with respect to $\boldsymbol{\sigma}$. These 'permutation checks' are done via a *permcheck* PIOP that uses multilinear extensions $\hat{\pi}, \hat{\sigma}$ to ensure that $\hat{\ell}(\boldsymbol{i}) = \hat{o}(\mathsf{bin}(\hat{\pi}(\boldsymbol{i})))$ and $\hat{r}(\boldsymbol{i}) = \hat{o}(\mathsf{bin}(\hat{\sigma}(\boldsymbol{i})))$, for every $\boldsymbol{i} \in \{0,1\}^{\log m}$.

Both these subPIOPs in turn depend on the PIOP for the *sumcheck* relation as noted in Fig. 1. We provide details about these PIOPs in Sections 2.5 and 2.6; as we explain there, existing prover algorithms for these PIOPs achieve the optimal linear time, but also require a linear amount of random-access space.

HyperPlonk's PC scheme. Chen et al. use the PST multilinear PC scheme [PST13]. We provide details

[6] HyperPlonk supports circuits that have higher arity gates that enforce complex logic, such as evaluating a degree-d polynomial over the gate inputs. We omit these details for brevity, but the HyperPlonk circuit relation definition in Definition 5.20 supports such 'custom gates'.

[7] For brevity, we omit a discussion of how HyperPlonk's representation handles inputs to the circuit and assume that every gate is an 'internal' gate with both left and right inputs. Our formal definition in Definition 5.20 lifts these restrictions and considers a standard circuit model.

[8] Formally, this preprocessing step is performed by the PIOP *indexer*, which we have not described here for brevity. Our formal definition of PIOPs in Section 5.1.1 describes the indexer's task in detail.

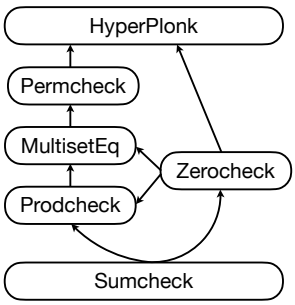

Figure 1: Components of the HyperPlonk PIOP. We make each component read-write streaming.

about this scheme in Section 2.8.1, but note here that while both the commitment and opening algorithms require only a linear number of field operations and a linear-sized multi-scalar multiplication, they also require a linear amount of space.

2.3 Read-write streams

As explained in the introduction, we will rely on a new model of 'read-write' streaming to construct SNARK provers that can efficiently prove large statements on devices with limited memory by relying on external memory. We describe our model below, including descriptions of read-write streams, algorithms that use these, efficiency measures for these algorithms, and how one can compose read-write streaming algorithms. We provide formal definitions in Section 4, and focus here on high-level intuition.

Read-write streams. A stream consists of a tape containing elements from some alphabet Σ, a *pointer* to a position on the tape, and a *mode* that is either 'read' or 'write'. If the stream's mode is 'read', then an algorithm can sequentially read elements from the stream (starting from the location pointed to by the pointer), while if its mode is 'write', the algorithm can sequentially write elements to the stream. More precisely, our model supports the following operations on streams:

- $\mathsf{init}(\cdot)$, which initializes a stream in write mode and allocates space for it,
- $\mathsf{restart}()$, which resets the pointer to the beginning of the stream,
- $\mathsf{read}()$, which reads a stream element from the location pointed to by the pointer (incrementing the pointer) if the stream is in read mode,
- $\mathsf{write}(\cdot)$, which writes the input value to the location pointed to by the pointer (incrementing the pointer) if the stream is in write mode.

Read-write streaming algorithms. A *read-write* (RW) streaming algorithm $\mathcal{A}$ has the following interface and behavior. It obtains its input via a stream $\mathbf{I}$ in read mode, and writes its output to a stream $\mathbf{O}$ in write mode. It has random-access to a small number of *registers*, and has read-write streaming access to a small number of intermediate read-write streams that are stored in a large external storage.[9] We denote an invocation of $\mathcal{A}$ on input $\mathbf{I}$ that produces output $\mathbf{O}$ as $\mathcal{A}(\mathbf{I}) \mapsto \mathbf{O}$.

Efficiency measures. The time complexity $t_{\mathcal{A}}$ of a read-write streaming algorithm $\mathcal{A}$ is defined as the total number of operations on registers plus the total number of stream operations performed during execution.

[9] Concretely, the number of registers and streams is at most logarithmic in the size of the input. Our model can support a logarithmic number of input streams without overhead by viewing the single input stream as the concatenation of these multiple streams, and by performing a single pass over it to copy each stream's contents to different intermediate streams. A similar statement holds for multiple output streams.

The *random-access* space complexity $m_{\mathcal{A}}$ of $\mathcal{A}$ is the maximum number of registers it uses during execution, and the *streaming* space complexity $s_{\mathcal{A}}$ is the maximum number of elements it stores in its intermediate read-write streams during execution.

Composing read-write streaming algorithms. A read-write streaming algorithm $\mathcal{A}$ can invoke another read-write streaming algorithm $\mathcal{B}$ as a subroutine. We call this process *composition* of read-write streaming algorithms, and at a high level, it works as follows. $\mathcal{A}$ allocates two intermediate streams: $\mathbf{I}$, the input stream of $\mathcal{B}$, and $\mathbf{O}$, the output stream of $\mathcal{B}$. $\mathcal{A}$ writes the input for $\mathcal{B}$ to $\mathbf{I}$, and then invokes $\mathcal{B}$ on $\mathbf{I}$. Once $\mathcal{B}$ terminates, $\mathcal{A}$ can read the output in $\mathbf{O}$ by re-interpreting it as a read stream, and can continue the rest of its computation. Composition is well-behaved with respect to time and space efficiency, as we show in Lemma 2.1.

Lemma 2.1 (informal)**.** *Let $\mathcal{A}$ be a read-write streaming algorithm that invokes another read-write streaming algorithm $\mathcal{B}$ as a subroutine L times. Denote by $t'_{\mathcal{A}}$, $m'_{\mathcal{A}}$, and $s'_{\mathcal{A}}$ the time complexity, random-access space complexity, and streaming space complexity, respectively, of $\mathcal{A}$ when* ignoring the cost of running $\mathcal{B}$*. Then the time complexity of $\mathcal{A}$ is $t_{\mathcal{A}} = t'_{\mathcal{A}} + L \cdot t_{\mathcal{B}}$, the random-access space complexity is $m_{\mathcal{A}} = m'_{\mathcal{A}} + m_{\mathcal{B}}$, and the streaming space complexity is $s_{\mathcal{A}} = s'_{\mathcal{A}} + s_{\mathcal{B}}$.*

In the foregoing lemma, the space complexity costs of $\mathcal{A}$ do not incur a factor of L multiplicative blowup with respect to the same costs for $\mathcal{B}$; $\mathcal{A}$ can reclaim the space used internally by each invocation of $\mathcal{B}$ after it finishes executing.

The notion of composition can be straightforwardly generalized to support multiple subroutines call, with the corresponding changes to total time and space complexities. For details, see Remark 4.5.

Remark 2.2 (applicability of read-write streaming)**.** While in this paper we primarily instantiate RW streams as files on disk storage, the RW streaming model is flexible enough to capture other settings where there is an imbalance between small (but fast) and large (but slow) memory. We provide some examples below, specialized for our use case of SNARK proving.

- Specialized hardware provers: there has been much recent industrial interest in hardware-accelerated proving (e.g., via GPUs, FPGAs, or custom ASICs). Unfortunately, such specialized hardware tends to have small on-device memory, which bottlenecks the size of the computations that can be proven. The CPU or other host device, on the other hand, has access to large CPU RAM, but moving data between the two types of memory is expensive. An RW streaming prover is a natural fit for this setting: the prover can run on the hardware device, and use the host device's memory as a large external storage. The streaming access pattern is entirely predictable, so the host device can efficiently pre-fetch data from its slow memory.
- Hardware-constrained devices: users of private cryptocurrencies cannot utilize hardware wallets to create private transactions without incurring privacy leakage. This is because these transactions contain SNARK proofs that cannot be proven in the wallet's limited memory, and so today wallets outsource proof generation to a host device, leaking sensitive transaction details in the process. However, if these SNARKs were equipped with an RW streaming prover, then hardware wallets could prove these SNARKs themselves by utilizing the host device's memory as external storage, and encrypting the contents of streams before sending them to the host device.

We leave it to future work to explore these applications of our RW streaming model.

2.4 SNARKs from RW streaming PIOPs and PC schemes

Let P be a RW streaming PIOP prover, and let PC be an RW streaming PC scheme. Then we can construct the RW streaming argument prover $\mathcal{P}$ in a manner analogous to the non-streaming case (Section 2.2): $\mathcal{P}$

receives on its input stream (s) any preprocessed polynomials and the NP witness and initializes P with these. In each round of the protocol, $\mathcal{P}$ invokes P with the latest verifier message to obtain a stream containing the current round polynomials. $\mathcal{P}$ then passes this stream to $\mathsf{PC.Commit}$ to obtain commitments to the round polynomials, and sends these to $\mathcal{V}$. At the end of the interaction, $\mathcal{P}$ invokes $\mathsf{PC.Open}$ with input streams comprised of both the preprocessed polynomials and the round polynomials produced during the interactive phase. $\mathcal{P}$ assembles the resulting opening proof and the commitments into the final SNARK proof and outputs this.

Efficiency. The argument prover $\mathcal{P}$ calls 3 subroutines: the PIOP Prover P, and the PC scheme algorithms $\mathsf{PC.Commit}$ and $\mathsf{PC.Open}$ Applying the version of Lemma 2.1 that supports multiple subroutines gives the following lemma:

Lemma 2.3. *Consider the following ingredients:*

- *A PIOP whose prover, on inputs of size N, requires time $t_{\mathsf{PIOP}}(N)$, random-access space $m_{\mathsf{PIOP}}(N)$, and streaming space $s_{\mathsf{PIOP}}(N)$. The PIOP prover produces c polynomials and the PIOP verifier queries o polynomials.*
- *A PC scheme whose algorithms have the following complexities on inputs of size N.*
  - $\mathsf{PC.Commit}$ *requires time $t_{\mathsf{C}}(N)$, random-access space $m_{\mathsf{C}}(N)$, and streaming space $s_{\mathsf{C}}(N)$.*
  - $\mathsf{PC.Open}$ *requires time $t_{\mathsf{O}}(N)$, random-access space, $m_{\mathsf{O}}(N)$, and streaming space $s_{\mathsf{O}}(N)$.*

Then there exists a read-write streaming argument whose prover $\mathcal{P}$ has the following efficiency properties:

- *Time complexity $t_{\mathcal{P}}(N) = t_{\mathsf{PIOP}}(N) + \mathsf{c} \cdot t_{\mathsf{C}}(N) + \mathsf{o} \cdot t_{\mathsf{O}}(N)$.*
- *Random-access space complexity $m_{\mathcal{P}}(N) = m_{\mathsf{PIOP}}(N) + m_{\mathsf{C}}(N) + m_{\mathsf{O}}(N)$.*
- *Streaming space complexity $s_{\mathcal{P}}(N) = s_{\mathsf{PIOP}}(N) + s_{\mathsf{C}}(N) + s_{\mathsf{O}}(N)$.*

Applying Lemma 2.3 to the RW streaming PIOP and PC scheme[10] that we construct in Sections 2.6 and 2.8 leads to the following corollary:

Corollary 2.4. *There exists a RW streaming argument prover $\mathcal{P}$ for circuit satisfiability that requires $O(N)$ cryptographic time, $O(\log N)$ random-access space, and $O(N)$ streaming space, where N is the size of the circuit.*

Proof. The prover for the PIOP of Section 2.6 requires $O(N)$ time, $O(\log N)$ random-access space, and $O(N)$ streaming space and the verifier queries $L = 6$ polynomials, while the commitment and opening algorithms of the PC scheme in Section 2.8 both require $O(N)$ cryptographic time, $O(\log N)$ random-access space, and $O(N)$ streaming space. □

We further note that the RW streaming model not only improves upon the asymptotic efficiency of state-of-the-art read-only streaming SNARKs [BCHO22; ZCLKZ24], but also allows us to make concrete optimizations that are not possible in the read-only setting; we detail these in Section 3.

Remark 2.5. The foregoing discussion does not specify how RW streams for the witness are produced. We show in Section 2.9 how to do this for the HyperPlonk relation in a way that allows random-access space complexity to scale with the space complexity of the computation, as opposed to the size of the corresponding circuit.

[10] We say that a PIOP or argument is read-write streaming if it has a read-write streaming prover algorithm, and that a PC scheme is read-write streaming if it has read-write streaming commitment and opening algorithms.

2.5 Read-write streaming sumcheck

A core component of many PIOPs (including ours) is the celebrated sumcheck protocol [LFKN92]. In this section, we show how to construct linear-time read-write streaming provers for the sumcheck protocol when specialized to (products of) multilinear polynomials. We begin below with an overview of existing (non-streaming) algorithms, and then show how to adapt these to the read-write streaming setting in Section 2.5.1.

Background: sumcheck protocol. At a high level, the sumcheck problem requires a prover P to convince a verifier V with only oracle access to an n-variate multilinear[11] polynomial $p(\boldsymbol{X})$, that its evaluations over the boolean hypercube $\{0,1\}^n \subset \mathbb{F}^n$ sum up to a claimed value σ. The sumcheck protocol [LFKN92] is a protocol for this problem that allows the verifier to run in time $O(n)$ (whereas the naive algorithm takes time 2^n). We describe a variant of the sumcheck protocol below that iterates from the last variable to the first; this deviation from the standard presentation (which iterates from the first variable to the last) is for ease of exposition of our results. We note that this variant, called the 'LSB' sumcheck because it starts the summation from the least significant bit of the input, has been used in prior streaming focused work as well [BHRRS20].

PIOP 1: SUMCHECK

$\langle \mathsf{P}(\boldsymbol{p}), \mathsf{V}([\![p]\!], \sigma)\rangle$:
For i in $[n, \ldots, 1]$:

1. P sends to V the univariate polynomial a_i defined as:

$$a_i(X_i) := \sum_{\boldsymbol{b} \in \{0,1\}^{i-1}} p(\boldsymbol{b}, X_i, r_{i+1}, \ldots, r_n)$$

2. If $i = n$, V sets $\sigma_i := \sigma$; else, it sets $\sigma_i := a_{i-1}(r_{i-1})$.
3. V checks if $a_i(0) + a_i(1) = \sigma_i$.
4. If $i \neq 1$: V samples $r_i \xleftarrow{\$} \mathbb{F}$ and sends it to P.
5. Else: V samples $r_1 \xleftarrow{\$} \mathbb{F}$ and accepts if $a_1(r_1) = p(r_1, \ldots, r_n)$.

Thaler [Tha13; XZZPS19] proposed a linear-time prover for the sumcheck PIOP. Below we recap the adaptation of this to the LSB setting, as we will show how to convert it to a streaming algorithm. We assume, like all prior work that relies on sumcheck [Set20; CBBZ23], that the prover P's input polynomial $p(\boldsymbol{X})$ is represented by $\boldsymbol{p}$, its evaluations over the boolean hypercube $\{0,1\}^n$ ordered lexicographically. We omit the details of the work of V since they are unchanged:

$\mathsf{P}(\boldsymbol{p})$:
1. Initialize a table $\boldsymbol{T} \leftarrow \boldsymbol{p}$.
2. For i in $[n, \ldots, 1]$:
3. Define $N_i := N/2^{n-i}$.
4. Compute evaluations $(e_i, o_i) \leftarrow \mathsf{Sum}(N_i, \boldsymbol{T})$.
5. Send $a_i(0) := e_i$ and $a_i(1) := o_i$ to V.
6. If $i \neq 1$:
7. Receive challenge $r_i \xleftarrow{\$} \mathbb{F}$ from V.
8. $\mathsf{FoldInPlace}(N_i, 1 - r_i, r_i, \boldsymbol{T})$.

$\mathsf{Sum}(N, \boldsymbol{T}) \to (\mathbb{F}, \mathbb{F})$:
1. Initialize $e \leftarrow 0$; $o \leftarrow 0$.
2. For j in $[1, \ldots, N/2]$:
3. $e \leftarrow e + T[2j]$; $o \leftarrow o + T[2j+1]$.
4. Output (e, o)

$\mathsf{FoldInPlace}(N, \alpha, \beta, \boldsymbol{T})$:
1. For j in $[1, \ldots, N/2]$:
2. $T[j] \leftarrow \alpha \cdot T[2j] + \beta \cdot T[2j+1]$.

The foregoing algorithm requires a table of size $N = 2^n$, and at the i-th iteration, it performs $N_i = N/2^{n-i}$ reads and writes from this table. This leads to the claimed time complexity of $O(N)$, but also, unfortunately, to a space complexity of $O(N)$.

[11] We describe how to handle more general polynomials in Section 5.2.

2.5.1 Linear-time RW streaming sumcheck

To improve this space complexity, we observe that in each iteration, all operations except Sum and Fold require $O(1)$ time and space. Thus, it suffices to design streaming algorithms for the latter two subroutines. Unfortunately, in the read-only streaming model, one cannot rely on the existence of the table $\boldsymbol{T}$ across iterations, and the prover naively recomputing it from scratch in each round leads to a time complexity of $O(N \log N)$. (But see Section 3 for ideas on how to reduce this to $O(N \log N / \log \log N)$). We further generalize these algorithms to kSum and kFoldInPlace, which can return more evaluations and handle multiple input streams (these are useful for the (d, ℓ)-sumprod sumcheck protocol, described in Section 5.2).

Our algorithm. In the read-write streaming model, on the other hand, we can initialize the table $\boldsymbol{T}$ in an intermediate read-write stream, and then read from and write to this stream to compute the folded tables. This leads to the following linear-time streaming prover for the sumcheck PIOP. (a) kSum reads from the table T in a streaming manner to compute $a_i(0)$ and $a_i(1)$, and (b) kFoldInPlace reads from *and* writes to the table T in a streaming manner using an intermediate stream. The resulting algorithm works as follows, where we highlight the streaming operations in blue:

P($\boldsymbol{p}$):
1. Initialize a read-stream $\boldsymbol{T} := \boldsymbol{p}$.
2. For i in $[n, \ldots, 1]$:
3. Define $N_i := N/2^{n-i}$.
4. Compute evaluations $(e_i, o_i) \leftarrow \mathsf{Sum}(N, \boldsymbol{T})$.
5. Send $a_i(0) := e_i$ & $a_i(1) := o_i$ to V.
6. If $i \neq 1$:
7. Receive challenge $r_i \xleftarrow{\$} \mathbb{F}$ from V.
8. FoldInPlace$(N_i, 1 - r_i, r_i, \boldsymbol{T})$.

Sum$(N, \boldsymbol{T}) \to (e, o)$:
1. Initialize $e \leftarrow 0$; $o \leftarrow 0$.
2. For j in $[1, \ldots, N/2]$:
3. $e \leftarrow e + \boldsymbol{T}$.read(); $o \leftarrow o + \boldsymbol{T}$.read().
4. Output (e, o).

FoldInPlace$(N, \alpha, \beta, \boldsymbol{T})$:
1. Initialize intermediate write-stream $\boldsymbol{I}$.
2. For j in $[1, \ldots, N/2]$:
3. $a \leftarrow \boldsymbol{T}$.read(); $b \leftarrow \boldsymbol{T}$.read().
4. $\boldsymbol{I}$.write$(\alpha \cdot a + \beta \cdot b)$.
5. Return $\boldsymbol{I}$.

It is straightforward to see that this algorithm preserves the desired time complexity of $O(N)$ and reduces the random-access space complexity to just $O(1)$. In Section 5.2, we extend this algorithm to support sums of products of multilinear polynomials (these are (d, ℓ)-sumprod polynomials in Table 1); this is necessary for building read-write streaming variants of the PIOPs in Section 2.6.

2.6 Read-write streaming PIOPs

With our RW streaming sumcheck PIOP in hand, we are now equipped to construct read-write streaming variants of the various PIOPs underlying the HyperPlonk PIOP.

2.6.1 Zerocheck

Given a polynomial p, the zerocheck PIOP [CBBZ23] aims to prove the following claim: "for all $\boldsymbol{x} \in \{0,1\}^n : p(\boldsymbol{x}) = 0$".

Reduction to sumcheck. The PIOP reduces the zerocheck claim to the sumcheck claim "$\sum_{\boldsymbol{b} \in \{0,1\}^{n-1}} p(\boldsymbol{b}) \cdot \mathrm{eq}(\boldsymbol{r}, \boldsymbol{b}) = 0$", where $\boldsymbol{r}$ is a random point sent by the verifier. (Recall that $\mathrm{eq}(\boldsymbol{r}, \boldsymbol{b}) = \prod_{i=1}^{n}((1 - r_i)(1 - b_i) + r_i b_i)$.)

PIOP 2: ZEROCHECK

$\langle \mathsf{P}(\boldsymbol{p}), \mathsf{V}([\![p(\boldsymbol{X})]\!]) \rangle$:

1. V samples $\boldsymbol{r} \xleftarrow{\$} \mathbb{F}^n$ and sends it to P.
2. P and V engage in a sumcheck of the product of $p(\boldsymbol{X})$ and $\mathrm{eq}(\boldsymbol{r}, \boldsymbol{X})$, with the target $\sigma = 0$.

Read-write streaming PIOP for zerocheck. We show how to create a streaming prover P for the zerocheck PIOP by designing a *read-only* streaming algorithm that generates the evaluations of the Lagrange basis polynomial, $[\mathrm{eq}(\boldsymbol{r}, \boldsymbol{b})]_{\boldsymbol{b} \in \{0,1\}^n}$. P then uses the read-write streaming sumcheck PIOP to prove the desired sumcheck claim.[12] The time and space complexity of this PIOP are determined by the cost of sumcheck and the cost of generating a stream for the evaluations of the Lagrange basis polynomials over $\{0,1\}^n$. The former requires $O(N)$ time with read-write streams as shown in Section 2.5.1, and we show below how to achieve the latter in $O(N)$ time with just read-only streams.

Computing the Lagrange basis polynomial. Given any $\boldsymbol{b} \in \{0,1\}^n$, computing $\mathrm{eq}(\boldsymbol{r}, \boldsymbol{b})$ takes $O(\log N)$ time. Therefore, at first glance it may seem like producing a stream of the evaluations of the Lagrange basis polynomial requires $O(N \log N)$ time in total. However, we can be more careful: to generate the stream $[\mathrm{eq}(\boldsymbol{r}, \boldsymbol{b})]_{\boldsymbol{b} \in \{0,1\}^n}$, we use two methods: we first run $\mathsf{Eq.Init}(\boldsymbol{r})$, which initializes a state given input $\boldsymbol{r}$ in $O(\log N)$ time, and then for each $\boldsymbol{b} \in \{0,1\}^n$ we run $\mathsf{Eq.Next}()$, which updates this state and outputs $\mathrm{eq}(\boldsymbol{r}, \boldsymbol{b})$, in $O(1)$ amortized time.

$\mathsf{Eq.Init}(\boldsymbol{r})$:

1. Define $S := \{i \in [n] : r_i \in \{0,1\}\}$.
2. Set $\mathsf{start} \leftarrow 0$ and $\mathsf{out} \leftarrow 0$.
3. Set $\mathsf{prod} \leftarrow \prod_{i \notin S}(1 - r_i)$. // (running evaluation of eq)
4. Set $\boldsymbol{b} \leftarrow [0]_{i=1}^n$ and store $\boldsymbol{r}$.

$\mathsf{Eq.Next}()$:

1. If $\mathsf{bin}(\boldsymbol{b}, i) = \mathsf{bin}(\boldsymbol{r}, i)$ for all $i \in S$: // ($O(1)$ time)
2. If $\mathsf{start} = 0$:
3. Set $\mathsf{start} \leftarrow 1$; $\mathsf{out} \leftarrow \mathsf{prod}$. // (first non-zero eval)
4. Else: For j in $[n, \ldots, 1] \setminus S$:
5. If $\mathsf{bin}(\boldsymbol{b}, j) = 0$: set $\mathsf{prod} \leftarrow \mathsf{prod} \cdot (1 - r_j)/r_j$.
6. Else: Set $\mathsf{out} \leftarrow \mathsf{prod} \leftarrow \mathsf{prod} \cdot r_j/(1 - r_j)$; Break.
7. Else: Set $\mathsf{out} \leftarrow 0$. // (there exists $i \in S$ s.t. $r_i \neq b_i$)
8. Increment $\boldsymbol{b} \leftarrow \mathsf{bin}(\mathsf{int}(\boldsymbol{b}) + 1)$. // ($O(1)$ time)
9. Output out.

We first analyze the case where $\boldsymbol{r} \in (\mathbb{F} \setminus \{0,1\})^n$, implying that $S = \varnothing$. $\mathsf{Eq.Init}$ initializes $\boldsymbol{b}$ with $[0]_{i=1}^n$ and prod with $\mathrm{eq}(\boldsymbol{r}, [0]_{i=1}^n) = \prod_{i=1}^n (1 - r_i)$, and $\mathsf{Eq.Next}$ then maintains the following invariant: $\mathsf{prod} = \prod_{i \in [n]}(r_i b_i + (1 - r_i)(1 - b_i))$, while incrementing $\boldsymbol{b}$.

One single call to $\mathsf{Eq.Next}$ might cost $O(n) = O(\log N)$ time, but if used N times, the calls have an amortized $O(1)$ cost. This is because each multiplication with $r_i/(1 - r_i)$ or $(1 - r_i)/r_i$ corresponds to flipping the i-th bit of $\boldsymbol{b}$. For each $i \in [n]$ this occurs at most 2^{n-i} times, and therefore the total number of flips is at most $2N$, which corresponds to the total number of multiplications required to compute the evaluations of eq. As an additional optimization, Steps 1 and 8 can be performed in $O(1)$ time by storing $\boldsymbol{b}$ as an integer and using bitwise operations.

For the other case where some $r_i \in \{0,1\}$, we cannot divide by r_i or $(1 - r_i)$, so we need to specially handle this. If $\boldsymbol{b}$ and $\boldsymbol{r}$ are equal on all the indices in S, then $\prod_{i \in S}((1 - r_i)(1 - b_i) + r_i b_i) = 1$, and therefore we simply return $\mathrm{eq}(\boldsymbol{r}, \boldsymbol{b}) = \prod_{i \notin S}((1 - r_i)(1 - b_i) + r_i b_i)$, which we iteratively update as required. If $\boldsymbol{b}$ and $\boldsymbol{r}$ are not equal on the indices in S, there exists an index $i \in S$ such that $r_i \neq b_i$, and therefore $\mathrm{eq}(\boldsymbol{r}, \boldsymbol{b})$ evaluates to 0.

[12] We provide a concrete optimization of this reduction via the *Lagrange sumcheck protocol* in Section 5.2.

Remark 2.6. As noted by Rothblum [Rot24], this approach was observed in an unpublished manuscript of Vu in 2013; however their description only provides a high-level overview of the technique, and does not provide a detailed algorithm. In particular, we handle the case where some of the r_i's are boolean, which is important for supporting batch opening of multiple polynomials at different points [CBBZ23].

2.6.2 Prodcheck

A prodcheck claim for two n-variate multilinear polynomials p and q says that the product of their evaluations over the hypercube $\prod_{\boldsymbol{x}\in\{0,1\}^n}(p(\boldsymbol{x})/q(\boldsymbol{x}))$ equals a claimed product $\sigma \in \mathbb{F}$. PIOPs for prodcheck claims are a fundamental tool for constructing PIOPs for more complex statements, including multiset-equality-check and permcheck.

Reduction to zerocheck. HyperPlonk relies on the prodcheck PIOP introduced by Setty and Lee [SL20]. In this PIOP, the prover P computes and sends to the verifier an $(n+1)$-variate polynomial ν defined as follows:

$$\nu(0, \boldsymbol{X}) = p(\boldsymbol{X})/q(\boldsymbol{X}) \qquad \nu(1, \boldsymbol{X}) = \nu(\boldsymbol{X}, 0) \cdot \nu(\boldsymbol{X}, 1)$$

P and V then engage in a zerocheck PIOP for the claims "$\nu(0, \boldsymbol{X}) \cdot q(\boldsymbol{X}) - p(\boldsymbol{X}) = 0$" and "$\nu(1, \boldsymbol{X}) - \nu(\boldsymbol{X}, 0) \cdot \nu(\boldsymbol{X}, 1) = 0$", which together ensure that ν is constructed correctly from p and q, and hence $\nu(1, 1, \ldots, 1, 0) = \prod_{\boldsymbol{x}\in\{0,1\}^n} p(\boldsymbol{x})/q(\boldsymbol{x})$. If this holds, then by construction of ν, the verifier V can simply query $\nu(1, 1, \ldots, 1, 0)$ and check that it equals σ to verify the prodcheck claim.

Because each zerocheck PIOP only requires $O(N)$ time and $O(\log N)$ random-access space, the only remaining challenge is to compute ν with the same complexity. The standard algorithm for computing ν does so recursively: for each $\boldsymbol{x} \in \{0,1\}^n$, let $\nu^{(0)}(\boldsymbol{x}) = p(\boldsymbol{x})/q(\boldsymbol{x})$, and, for each $i \in \{0, \ldots, n-1\}$ and each $\boldsymbol{x} \in \{0,1\}^{n-i}$, set $\nu^{(i+1)}(\boldsymbol{x}) = \nu^{(i)}(1, \boldsymbol{x}) \cdot \nu^{(i)}(0, \boldsymbol{x})$. Then ν is the concatenation of the $\nu^{(i)}$'s. With linear space, this computation can be done in $O(N)$ time. What about when we have only logarithmic space?

Barrier to read-only streaming. A streaming variant of the foregoing algorithm would only be able to produce a stream for ν by producing, in order, streams for $\nu^{(0)}, \nu^{(1)}, \ldots, \nu^{(n-1)}$. While doing so for $\nu^{(0)}$ is straightforward, producing the stream for $\nu^{(i+1)}$ requires computing the product of two evaluations of $\nu^{(i)}$, which in turn would either require recomputing $\nu^{(i)}$ (and hence every $\nu^{(j)}$ for $j < i$), or storing it in memory. Since the latter is not an option in the read-only streaming model, we are left with former approach which requires $O(N \log N)$ time overall.

Remark 2.7. There is an alternative approach that outputs a stream containing the entries of the $\nu^{(i)}$'s, but in an interleaved form. The idea is to make explicit the tree induced by the recursion, and then, instead of exploring the tree in a breadth-first manner (i.e., producing the leaves $\nu^{(0)}$, then the layer above it, and so on), we explore it in a depth-first manner. This approach works and achieves $O(N)$ time with $O(\log N)$ space, but it produces ν's evaluations in an interleaved manner, which is incompatible with our zerocheck and sumcheck PIOPs, as the latter expect lexicographically ordered evaluations.

Producing ν using RW streams. In the read-write setting, we *can* store the evaluations of $\nu^{(i)}$ on disk, and therefore we can produce the stream for ν in $O(N)$ time and $O(\log N)$ random-access space. We defer to Section 5.3 a detailed description of this algorithm and the overall PIOP. We describe the algorithm below.

$\mathsf{P}(\boldsymbol{p}, \boldsymbol{q})$:
1. Initialize a read-stream $\boldsymbol{\nu}^{(0)} := \boldsymbol{p}/\boldsymbol{q}$.
2. For i in $[1, \ldots, n-1]$:
3. Initialize a write-stream $\boldsymbol{\nu}^{(i)}$ of size 2^{n-i}.
4. For j in $[1, \ldots, 2^{n-i}]$:
5. $a \leftarrow \boldsymbol{\nu}^{(i-1)}.\mathsf{read}()$; $b \leftarrow \boldsymbol{\nu}^{(i-1)}.\mathsf{read}()$.
6. $\boldsymbol{\nu}^{(i)}.\mathsf{write}(a \cdot b)$.
7. Re-initialize $\boldsymbol{\nu}^{(i)}$ as a read-stream.
8. Return read-stream $\boldsymbol{\nu} := (\boldsymbol{\nu}^{(0)} \mid\mid \boldsymbol{\nu}^{(1)} \mid\mid \ldots \mid\mid \boldsymbol{\nu}^{(n-1)})$.

Further details are provided in Section 5.3.2.

2.6.3 Multiset-equality-check

Given two n-variate multilinear polynomials p and q, the multiset-equality-check PIOP aims to prove that the evaluation tables over the boolean hypercube of p and q are equal as multisets. More precisely, it aims to prove the following claim: "$\{\!\{p(\boldsymbol{x}) : \boldsymbol{x} \in \{0,1\}^n\}\!\} = \{\!\{q(\boldsymbol{x}) : \boldsymbol{x} \in \{0,1\}^n\}\!\}$".

Reduction to prodcheck. This claim is proven via a reduction to a prodcheck claim: V sends a random challenge $\alpha \in \mathbb{F}$ to P, and then P and V engage in a prodcheck PIOP for the claim that $\prod_{\boldsymbol{x} \in \{0,1\}^n} (p(\boldsymbol{x}) + \alpha)/(q(\boldsymbol{x}) + \alpha) = 1$.

Streaming PIOP. The prodcheck PIOP is invoked on the polynomials $(p(\boldsymbol{x}) + \alpha)_{\boldsymbol{x} \in \{0,1\}^n}$ and $(q(\boldsymbol{x}) + \alpha)_{\boldsymbol{x} \in \{0,1\}^n}$. Streaming access to these polynomials can be easily simulated with streaming access to p and q, and therefore the reduction is in fact read-only streaming. In order to be more concretely efficient and avoid disk accesses, we leverage read-only streaming reductions whenever possible. Therefore, the reduction requires $O(N)$ time, and only $O(1)$ additional space (both streaming and random-access). Further details are provided in Section 5.3.3.

Multiset-equality-check with log derivatives. We also present an RW streaming version of this PIOP that relies on the PIOP of Haböck [Hab22], which reduces a multiset-equality instance to sumcheck and zerocheck instances directly. We also discuss some concrete optimizations for this PIOP through a reduction to a *sumcheck for rational functions* as described in Appendix A.

2.6.4 Permcheck

Given two n-variate multilinear polynomials p and q, and a permutation $\pi : \{0,1\}^n \to \{0,1\}^n$, the permcheck PIOP aims to prove the following claim: "$p(\boldsymbol{x}) = q(\pi(\boldsymbol{x}))$ for all $\boldsymbol{x} \in \{0,1\}^n$".

Reduction to multiset-equality-check. This claim is proven via through a reduction to a multiset-equality-check claim: V sends a random challenge $\beta \in \mathbb{F}$ to P, and then P and V engage in a multiset-equality-check PIOP for the claim $\{p(\boldsymbol{x}) + \beta \cdot \mathsf{int}(\boldsymbol{x})\}_{\boldsymbol{x} \in \{0,1\}^n} = \{q(\boldsymbol{x}) + \beta \cdot \mathsf{int}(\pi(\boldsymbol{x}))\}_{\boldsymbol{x} \in \{0,1\}^n}$.[13]

Streaming PIOP. Similar to the reduction from multiset-equality-check to prodcheck, the read-streams for both $\{p(\boldsymbol{x}) + \beta \cdot \mathsf{int}(\boldsymbol{x})\}_{\boldsymbol{x} \in \{0,1\}^n}$ and $\{p(\boldsymbol{x}) + \beta \cdot \mathsf{int}(\boldsymbol{x})\}_{\boldsymbol{x} \in \{0,1\}^n}$ can be simulated efficiently in a read-only streaming manner. For any $\boldsymbol{x} \in \{0,1\}^n$, P can compute $p(\boldsymbol{x}) + \beta \cdot \mathsf{int}(\boldsymbol{x})$ and $q(\boldsymbol{x}) + \beta \cdot \mathsf{int}(\pi(\boldsymbol{x}))$ in constant time and space since it has access to $p(\boldsymbol{x}), q(\boldsymbol{x}), \mathsf{int}(\boldsymbol{x}), \mathsf{int}(\pi(\boldsymbol{x}))$. Therefore, both streams can be

[13] The stream of $\mathsf{int}(\boldsymbol{x})$ can be easily simulated because it consists of integers from 0 to $N-1$. P also receives $\hat{\pi}$ as input, which is the multilinear extension of $\tilde{\pi}$, where $\tilde{\pi} : \{0,1\}^n \to \mathbb{F}$ is obtained by casting the output of π to an element of $\mathbb{F}$.

produced in $O(N)$ time, and only $O(1)$ additional space (both streaming and random-access). Further details are provided in Section 5.3.4.

2.7 PIOP for HyperPlonk

As described in Section 2.2, the HyperPlonk PIOP reduces circuit satisfiability to a permcheck claim and a zerocheck claim. We show in Section 5.3 how to construct a read-write PIOP that performs this reduction. Our PIOP requires $O(N)$ time, $O(\log N)$ random-access space, and $O(N)$ streaming space complexity and satisfies Corollary 2.4.

The time and space complexity of the aforementioned PIOPs are summarized in Table 1.

PIOP	time	space	
		random-access	streaming
multilinear sumcheck	$O(N)$	$O(1)$	$O(N)$
(d, ℓ)-sumprod sumcheck	$O(d\ell N)$	$O(d + \ell)$	$O(dN)$
(d, ℓ)-sumprod zerocheck	$O(d\ell N)$	$O(d + \ell + \log N)$	$O(dN)$
prodcheck	$O(N)$	$O(\log N)$	$O(N)$
multiset-equality	$O(N)$	$O(\log N)$	$O(N)$
permcheck	$O(N)$	$O(\log N)$	$O(N)$
HyperPlonk	$O(N)$	$O(\log N)$	$O(N)$

Table 1: Efficiency of our read-write streaming PIOPs. All complexities are in terms of number of field elements/operations. The verifier time, query complexity and soundness error are the same as their non-streaming counterparts in HyperPlonk [CBBZ23].

2.8 Read-write streaming polynomial commitments

We now show how to construct read-write streaming variants of a number of popular polynomial commitment schemes for multilinear polynomials. Our constructions preserve the cryptographic linear-time complexity of the prover, while reducing the random-access space to just $O(\log N)$. Our results are summarized in Table 2.

scheme	SRS size	time		streaming space		check time	proof size
		commit	open	commit	open		
PST13 [PST13]	$O(N)$	$O(N)$	$O(N)$	$O(1)$	$O(N)$	$O(\log N)$	$O(\log N)$
Hyrax [WTSTW18]	$O(1)$	$O(N)$	$O(N)$	$O(1)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N})$
Multilinear Halo [BGH19; Set20]	$O(1)$	$O(N)$	$O(N)$	$O(1)$	$O(N)$	$O(N)$	$O(\log N)$
BMMTV21 [BMMTV21]	$O(\sqrt{N})$	$O(N)$	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\log N)$	$O(\log N)$
Dory [Lee21]	$O(\sqrt{N})$	$O(N)$	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\log N)$	$O(\log N)$

Table 2: Efficiency of our read-write streaming polynomial commitment schemes for n-variate multilinear polynomials, where $N = 2^n$. All sizes are specified in number of group elements, and all time complexities in number of group operations.

We now provide high-level overviews of the techniques used to obtain the results in Table 2. We begin in Section 2.8.1 by describing the read-write streaming variant of the scheme of Papamanthou, Shi, and Tamassia (PST) [PST13]. Then, in Section 2.8.2, we show how to obtain a cryptographic linear-time RW

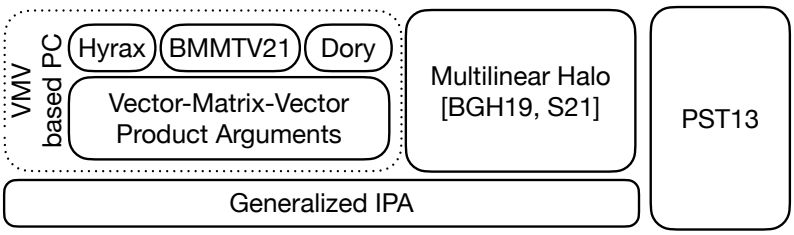

Figure 2: Our read-write streaming PC schemes.

streaming prover for generalized inner product arguments [BMMTV21], and then show how to use the latter to construct RW streaming variants of the schemes of Bowe et al. [BGH19] (Section 2.8.3), and of Wahby et al. [WTSTW18], Bünz et al. [BMMTV21] and of Lee [Lee21] (Section 2.8.4). In all cases, we focus on the commitment and opening algorithms, as these are the ones that are invoked by the RW streaming SNARK prover.

Building block: multi-scalar multiplication. A key component of all the aforementioned constructions is a *multi-scalar multiplication* (MSM) operation, which computes $\sum_{i=1}^{N} a_i \cdot G_i$ for given scalars (field elements) $a_1, \ldots, a_N$ and group elements $G_1, \ldots, G_N$. Given streaming access to $\boldsymbol{a}$ and $\boldsymbol{G}$, clearly this operation can be performed in a read-only streaming manner in cryptographic linear time: simply stream through $\boldsymbol{a}$ and $\boldsymbol{G}$ in parallel, and compute the sum incrementally.

2.8.1 Read-write streaming variant of PST

The PST scheme [PST13] extends the ideas of the KZG scheme [KZG10] from univariate polynomials to multilinear polynomials. We start by recalling a version of the PST scheme presented in Libra [XZZPS19].

Setup. Setup samples a bilinear group $\langle \mathsf{group} \rangle = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G, H, e) \leftarrow \mathsf{SampleGrp}(1^\lambda)$, and outputs a commitment key of the form $\mathsf{ck} := ([\mathsf{ck}_j]_{j=0}^{n})$, where $\mathsf{ck}_j := [\mathrm{eq}_j(\boldsymbol{\alpha}_{[n-j+1:n]}, \boldsymbol{i}) \cdot G]_{\boldsymbol{i} \in \{0,1\}^j}$,[14] and $\boldsymbol{\alpha} \xleftarrow{\$} \mathbb{F}^n$ is a randomly sampled field vector. Each ck_j enables committing to a j-variate multilinear polynomial.[15]

Commit. Given an n-variate multilinear polynomial p represented as its evaluations over the boolean hypercube $\boldsymbol{p} := [p(\mathsf{bin}(i))]_{i=0}^{2^n-1}$, Commit computes the commitment $C := \langle \boldsymbol{p}, \mathsf{ck}_n \rangle$ via a read-write streaming MSM.

Opening. Computing an evaluation proof is more challenging. For an n-variate multilinear polynomial $p(X_1, \ldots, X_n)$ and an evaluation point $\boldsymbol{z} \in \mathbb{F}^n$, PST relies on the fact that $p(\boldsymbol{z}) = v$ if and only if there exist polynomials $q_1(X_2, \ldots, X_n), q_2(X_3, \ldots, X_n), \ldots, q_{n-1}(X_n), q_n$ such that

$$p(\boldsymbol{X}) - v = \sum_{i=1}^{n} q_i(X_{i+1}, \ldots, X_n) \cdot (X_i - z_i) \ .$$

PST leverages this fact as follows: to prove that $p(\boldsymbol{z}) = v$, Open computes commitments $\pi_1, \ldots, \pi_n$ to the n 'witness' polynomials $q_1, \ldots, q_n$ with respect to keys $\mathsf{ck}_{n-1}, \ldots, \mathsf{ck}_0$ respectively. Since these polynomials are respectively of sizes $2^{n-1}, 2^{n-2}, \ldots, 1$, these commitments can be computed in overall cryptographic linear-time, and so we are left to reason about the time complexity of computing the witness polynomials themselves.

[14] Given a vector $\boldsymbol{a} = [a_i]_{i=0}^{n}$, we use $\boldsymbol{a}_{[j:k]}$ to denote $[a_i]_{i=j}^{k}$.

[15] To avoid any ambiguity, we explicitly define $\mathsf{ck}_0 := G$.

Zhang et al. [ZGKPP18, Appendix G] demonstrate a linear-time algorithm for computing these polynomials that relies on the following decomposition of the multilinear polynomial p:

$$\begin{aligned} p(X_1, \ldots, X_n) &= g(X_2, \ldots, X_n) + X_1 \cdot h(X_2, \ldots, X_n) \\ &= (g(X_2, \ldots, X_n) + z_1 \cdot h(X_2, \ldots, X_n)) \\ &\quad + (X_1 - z_1) \cdot h(X_2, \ldots, X_n) \\ &:= r_1(X_2, \ldots, X_n) + (X_1 - z_1) \cdot q_1(X_2, \ldots, X_n) \end{aligned}$$

With q_1 in hand, Open proceeds to compute q_2 by recursively invoking the foregoing decomposition on the 'remainder' polynomial $r_1(X_2, \ldots, X_n)$. Then for each $j \in \{2, 3, \ldots, n\}$, Open recursively invokes the foregoing decomposition on the 'remainder' polynomial $r_{j-1}(X_j, \ldots, X_n)$ to obtain the witness polynomial $q_j(X_{j+1}, \ldots, X_n)$ and the next remainder polynomial $r_j(X_{j+1}, \ldots, X_n)$. Zhang et al. [ZGKPP18] show how to compute this decomposition in linear time.

Limitations of read-only streaming. While Commit is computable in a read-only streaming manner, it is more difficult to realize a read-only streaming variant of Open that runs in cryptographic linear-time.

In more detail, a naive read-only streaming adaptation of the algorithm of Zhang et al. would require $O(N \log N)$ time because, for each $i \in [\log N]$, it would have to compute the evaluations $\boldsymbol{q}_i$ of the polynomial $q_i(\boldsymbol{X})$ from scratch, which would take $O(N)$ time for each i.

We can avoid this by using ideas similar to those in our streaming prodcheck algorithm in Section 2.6.2, and changing the order in which we produce the evaluations $\boldsymbol{q}_i$. Namely, we can view the computation that produces these polynomials as implicitly traversing a binary tree. The naive adaptation of the algorithm of Zhang et al. would traverse this tree in a level-by-level order manner, computing all the evaluations of a particular $\boldsymbol{q}_i$ before moving on to the next one. We could instead traverse it in a depth-first manner, thus not having to compute each element of each $\boldsymbol{q}_i$ from scratch, and only requiring $O(N)$ time overall. The downside to this approach is that since the evaluations of the witness polynomial are now produced in an interleaved manner, committing to these in a read-only streaming manner would require a similarly interleaved commitment key, which would double the size of the overall commitment key.

Achieving RW streaming in cryptographic linear-time. We instead describe a simpler, cryptographic linear-time, RW streaming Open algorithm, that avoids interleaving the commitment key. Our construction follows from the observation that each step of the algorithm of Zhang et al. closely parallels the kFoldInPlace step of the sumcheck protocol. Thus we can apply similar techniques to those used in Section 2.5 and leverage intermediate work streams to obtain a RW streaming version of Open.

Let $\boldsymbol{p}$, $\boldsymbol{g}$ and $\boldsymbol{h}$ be vectors consisting of evaluations (over the corresponding boolean hypercubes) of $p(X_1, \ldots, X_n)$, $g(X_2, \ldots, X_n)$ and $h(X_2, \ldots, X_n)$ respectively. Then, given a point $\boldsymbol{i} \in \{0,1\}^n$ with $i_1 = 0$, clearly $p(0, i_2, \ldots, i_n) = g(i_2, \ldots, i_n)$. Similarly, if $i_1 = 1$, then $p(1, i_2, \ldots, i_n) = g(i_2, \ldots, i_n) + h(i_2, \ldots, i_n)$, implying that $h(i_2, \ldots, i_n) = p(1, i_2, \ldots, i_n) - p(0, i_2, \ldots, i_n)$, and so we can set $\boldsymbol{g} := \boldsymbol{p}_L$ and $\boldsymbol{h} := \boldsymbol{p}_R - \boldsymbol{g}$, where $\boldsymbol{p}_L$ and $\boldsymbol{p}_R$ are the left and right halves of the evaluations of p, respectively.

Given streaming access to $\boldsymbol{p}$, clearly the prover can write $\boldsymbol{g}$ and $\boldsymbol{h}$ onto intermediate write streams. The prover then sets $\boldsymbol{q}_1 := \boldsymbol{h}$ and commits to it (we show how to do this in the next paragraph). Finally, the prover computes $\boldsymbol{r}_1 := \boldsymbol{g} + z_1 \cdot \boldsymbol{h}$ using the streams containing $\boldsymbol{g}$ and $\boldsymbol{h}$ and continues to iteratively obtain $\boldsymbol{q}_2, \ldots, \boldsymbol{q}_n$.

The streaming prover needs to compute the commitment C_j to the witness polynomial $q_j(X_{j+1}, \ldots, X_n)$ for all $j \in [n]$. The commitment is defined as follows: $C_j := \sum_{\boldsymbol{i} \in \{0,1\}^{n-j}} q_j(\boldsymbol{i}) \cdot \mathrm{eq}_{n-j}(\boldsymbol{\alpha}_{[j+1,n]}, \boldsymbol{i}) \cdot G$. This can be computed in a streaming manner straightforwardly, given streaming access to ck_{n-j} and $\boldsymbol{q}_j$. The full details of our RW streaming construction are presented in Section 6.2.1.

2.8.2 Building block: read-write streaming inner product arguments

A core building block for many polynomial commitment schemes [BCCGP16; WTSTW18; Lee21; BMMTV21; BGH19; BCMS20] is an inner product argument (IPA) [BCCGP16; BBBPWM18] or its generalization [LMR19; BMMTV21]. Thus, if we wish to design RW streaming variants of these polynomial commitments, it is essential to first design RW streaming variants of these IPAs.

Generalized Inner Product Arguments. We construct a RW streaming prover that requires only logarithmic random access space for the generalized inner product argument (GIPA) of [BMMTV21]. For the purposes of this section, an inner product argument allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that the inner product $\langle \boldsymbol{w}, \boldsymbol{x} \rangle = v$, where $\boldsymbol{w} \in \mathbb{F}^N$ is a private vector committed to in a Pedersen commitment C and $\boldsymbol{x} \in \mathbb{F}^N$ is a public vector shared by both $\mathcal{P}$ and $\mathcal{V}$.[16]

At a high level, the construction works as follows: $\mathcal{P}$ and $\mathcal{V}$ both receive as input (1) a commitment key consisting of a vector of group generators $\boldsymbol{G} \in \mathbb{G}^N$, (2) the Pedersen commitment $C := \sum_{i=1}^{n} w_i \cdot G_i = \langle \boldsymbol{w}, \boldsymbol{G} \rangle$, (3) the public vector $\boldsymbol{x}$, and (4) the claimed inner product value v. $\mathcal{P}$ additionally receives as input the private witness $\boldsymbol{w}$.

$\mathcal{P}$ and $\mathcal{V}$ engage in the following interactive protocol that reduces verifying the validity of the above claim to verifying the validity of a claim of half the size. (Recall that, for a vector $\boldsymbol{a}$, $\boldsymbol{a}_E$ is the vector containing the elements of $\boldsymbol{a}$ that appear at even indices, and $\boldsymbol{a}_O$ is the vector containing elements at odd indices.)

1. $\mathcal{P}$ computes $C_+ := \langle \boldsymbol{w}_E, \boldsymbol{G}_O \rangle$ and $C_- := \langle \boldsymbol{w}_O, \boldsymbol{G}_E \rangle$.
2. $\mathcal{P}$ computes $v_+ := \langle \boldsymbol{w}_E, \boldsymbol{x}_O \rangle$ and $v_- := \langle \boldsymbol{w}_O, \boldsymbol{x}_E \rangle$.
3. $\mathcal{P}$ sends the cross-terms C_+, C_-, v_+, v_- to $\mathcal{V}$.
4. $\mathcal{V}$ samples $\alpha \xleftarrow{\$} \mathbb{F}$ and sends it to $\mathcal{P}$.
5. $\mathcal{P}$ sets $\boldsymbol{w}' := \alpha \boldsymbol{w}_E + \boldsymbol{w}_O$.
6. $\mathcal{P}$ and $\mathcal{V}$ set

$$\begin{aligned} \boldsymbol{G}' &:= \alpha^{-1} \boldsymbol{G}_E + \boldsymbol{G}_O, \\ C' &:= C + \alpha C_+ + \alpha^{-1} C_-, \\ \boldsymbol{x}' &:= \alpha^{-1} \boldsymbol{x}_E + \boldsymbol{x}_O, \\ v' &:= v + \alpha v_+ + \alpha^{-1} v_-. \end{aligned}$$

This reduction guarantees, except with negligible probability over the choice of α, that if $C' = \langle \boldsymbol{w}', \boldsymbol{G}' \rangle$ and $v' = \langle \boldsymbol{w}', \boldsymbol{x}' \rangle$, then it must have been the case that $C = \langle \boldsymbol{w}, \boldsymbol{G} \rangle$ and $v = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$. Clearly the prover in this reduction runs in cryptographic linear-time as it only performs inner-products and linear combinations.

In the full GIPA, $\mathcal{P}$ and $\mathcal{V}$ run this interactive reduction recursively for $\log N$ rounds until $\boldsymbol{w}'$ is of length $O(1)$, at which point $\mathcal{P}$ can send $\boldsymbol{w}'$ to $\mathcal{V}$ in the clear and $\mathcal{V}$ can check that $C' = \langle \boldsymbol{w}', \boldsymbol{G}' \rangle$ and $v' = \langle \boldsymbol{w}', \boldsymbol{x}' \rangle$. Since the vectors in each round are respectively of sizes $2^{n-1}, 2^{n-2}, \ldots, 1$, the whole protocol can be executed in cryptographic linear-time.

Additionally, since this protocol is public-coin, it can be turned non-interactive via the Fiat–Shamir transform [FS86].

Barrier to read-only streaming. In each of the $\log N$ rounds, $\mathcal{P}$ has to compute and send the cross terms C_+, C_-, v_+, v_- to $\mathcal{V}$. This requires access to the intermediate vectors $\boldsymbol{G}'$, $\boldsymbol{x}'$ and $\boldsymbol{w}'$, and the computation of these closely resembles the Fold step of the sumcheck protocol (see Section 2.5). Indeed, just like in sumcheck, the GIPA prover 'folds' the vectors $\boldsymbol{G}$, $\boldsymbol{x}$ and $\boldsymbol{w}$ in half with respect to the challenge α. As a result, a read-only streaming version of the GIPA prover $\mathcal{P}$ would encounter the same bottleneck as the

[16] Generalized inner product arguments consider statements of a more general form: $\boldsymbol{w}, \boldsymbol{x}$ do not need to be vectors over $\mathbb{F}$. Further details can be found in Appendix B.

read-only streaming sumcheck prover: it would need to recompute the 'state' $\boldsymbol{G}'$, $\boldsymbol{x}'$ and $\boldsymbol{w}'$ for each round from scratch, which takes $O(N)$ work for each of the $\log N$ rounds. This is the approach taken by Block et al. [BHRRS20], and as a result they suffer from a $\log N$ overhead on the prover time.

Achieving RW streaming in cryptographic linear-time. We present a direct construction for a RW streaming GIPA prover that achieves cryptographic linear-time and logarithmic random-access space. Given streaming access to $\boldsymbol{G}$, $\boldsymbol{x}$ and $\boldsymbol{w}$, the RW streaming prover can compute the cross-terms as well as compute and write out $\boldsymbol{G}'$, $\boldsymbol{x}'$ and $\boldsymbol{w}'$ to an intermediate stream in a straightforward way using $O(N)$ group and field operations. It then iteratively applies the streaming reduction on these intermediate streams $\log N - 1$ more times, where the size of each stream is halved, resulting in a total of $O(N)$ operations. The prover only requires $O(1)$ random-access space to keep track of the pointers for the input and intermediate streams.

We note that there is another, more indirect path to obtaining a RW streaming GIPA prover: Bootle, Chiesa, and Sotiraki [BCS21] show how to interpret IPAs as *sumcheck arguments* where the prover's algorithm closely resembles the sumcheck prover's algorithm. We can exploit this interpretation by applying our techniques from Section 2.5 to obtain a RW streaming GIPA prover with the same asymptotic efficiency as our direct construction.

2.8.3 PC schemes directly from inner product arguments

Evaluating a multilinear polynomial $p(\boldsymbol{X})$ at a point $\boldsymbol{z}$ is equivalent to computing the inner product $\langle \boldsymbol{p}, \mathrm{eq}_{\boldsymbol{z}} \rangle = \sum_{\boldsymbol{b} \in \{0,1\}^n} p(\boldsymbol{b}) \mathrm{eq}(\mathbf{z}, \boldsymbol{b})$, where $\boldsymbol{p} := [p(\mathsf{bin}(i))]_{i=0}^{2^n-1}$ is the evaluation of the polynomial over its corresponding boolean hypercube and $\mathrm{eq}_{\boldsymbol{z}} := [\mathrm{eq}(\boldsymbol{z}, \mathsf{bin}(i))]_{i=0}^{2^n-1}$ (i.e. the vector consisting of the $\boldsymbol{i}$-th Lagrange polynomials evaluated at $\boldsymbol{z}$, for all $\boldsymbol{i} \in \{0,1\}^n$). Thus, an IPA immediately implies a polynomial commitment scheme:

- PC.Setup: Sample the commitment key $\boldsymbol{G} \xleftarrow{\$} \mathbb{G}^N$.
- PC.Commit: Compute a Pedersen-style commitment to the polynomial $\boldsymbol{p}$ as $C := \langle \boldsymbol{p}, \boldsymbol{G} \rangle$.
- PC.Open: Use the GIPA to prove that $p(\boldsymbol{z}) = v$ by proving $C = \langle \boldsymbol{p}, \boldsymbol{G} \rangle$ and $v = \langle \boldsymbol{p}, \mathrm{eq}_{\boldsymbol{z}} \rangle$.

Clearly, the RW streaming IPA from Section 2.8.2 implies RW streaming versions of the commitment and opening algorithms, with the only subtlety arising in the computation of $\mathrm{eq}_{\boldsymbol{z}}$, which we demonstrated how to compute in a read-only streaming manner in Section 2.6.1.

This is precisely the PC scheme from [BGH19]. It requires linear verifier time and a linear-sized commitment key. We describe next RW streaming algorithms for PC schemes that avoid these drawbacks.

2.8.4 PC schemes from VMV products

Background: polynomials as matrices. Wahby et al. [WTSTW18] proposed an alternate recipe for constructing polynomial commitment schemes from inner product arguments, where the size of the commitment key is sublinear in the size of the polynomial being committed to. At a high level, the recipe proceeds by viewing an n-variate multilinear polynomial $p(\boldsymbol{X})$, represented by its evaluation over the boolean hypercube $\boldsymbol{p}$, as a matrix $\mathbf{M} \in \mathbb{F}^{\sqrt{N} \times \sqrt{N}}$ defined as follows:

$$M_{ij} := p_k \text{ for } k = i \cdot 2^m + j \quad ,$$

where $N := 2^n$ and $m := n/2$.

They observe that evaluating $p(X_1, \ldots, X_n)$ at a point $\boldsymbol{z} = (z_1, z_2, \ldots, z_n)$ is equivalent to computing the vector-matrix-vector (VMV) product $\boldsymbol{\ell}^\top \mathbf{M} \boldsymbol{r}$, where $\boldsymbol{\ell} := [\mathrm{eq}(\boldsymbol{z}_L, \mathsf{bin}_n(i))]_{i=0}^{\sqrt{N}-1} = \otimes_{i=1}^{m} (1 - z_i, z_i)$ and

$\boldsymbol{r} := [\mathrm{eq}(\boldsymbol{z}_R, \mathsf{bin}_n(i))]_{i=0}^{\sqrt{N}-1} = \otimes_{i=m+1}^{n}(1-z_i, z_i)$.[17]

This can be illustrated as follows: consider a multilinear polynomial in four variables $p(X_1, X_2, X_4, X_4)$. Then for any $\mathbf{z} \in \mathbb{F}^4$ we can write $p(\mathbf{z}) = \sum_{\boldsymbol{b}\in\{0,1\}^4} p(\boldsymbol{b}) \prod_{i\in[4]}(1-z_i)^{1-b_i} z_i^{b_i}$. Now consider the following vectors:

$$\boldsymbol{\ell} = (1-z_1, z_1) \otimes (1-z_2, z_2) = ((1-z_1)(1-z_2), (1-z_1)z_2, z_1(1-z_2), z_1 z_2),$$
$$\boldsymbol{r} = (1-z_3, z_3) \otimes (1-z_4, z_4) = ((1-z_3)(1-z_4), (1-z_3)z_4, z_3(1-z_4), z_3 z_4),$$
$$\mathbf{M} = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 & p_7 \\ p_8 & p_9 & p_{10} & p_{11} \\ p_{12} & p_{13} & p_{14} & p_{15} \end{bmatrix} = \begin{bmatrix} \boldsymbol{M}_0 \\ \boldsymbol{M}_1 \\ \boldsymbol{M}_2 \\ \boldsymbol{M}_3 \end{bmatrix}$$

It can be inspected that $\boldsymbol{\ell}^\top \mathbf{M} \boldsymbol{r} = \sum_{\boldsymbol{b}\in\{0,1\}^4} p(\boldsymbol{b}) \prod_{i\in[4]}(1-z_i)^{1-b_i} z_i^{b_i} = p(\mathbf{z})$. This idea leads to the following blueprint for building polynomial commitment schemes:

Setup. Sample the commitment key $\boldsymbol{G} \xleftarrow{\$} \mathbb{G}^{\sqrt{N}}$.

Commit. To commit to the polynomial $\boldsymbol{p}$, Commit commits to the corresponding matrix $\mathbf{M}$ by Pedersen committing to each row $\boldsymbol{M}_i$ as $C_i := \langle \boldsymbol{M}_i, \boldsymbol{G} \rangle$, obtaining $\sqrt{N}$ row commitments. Some schemes like Hyrax [WTSTW18] directly use these row commitments $\boldsymbol{C} := [C_i]_{i=0}^{\sqrt{N}-1}$ as a $\sqrt{N}$-sized commitment to the polynomial. Other schemes, like Dory [Lee21] and that of Bünz et al. (BMMTV21) [BMMTV21], further commit to these row commitments (which are group elements) via a structure-preserving commitment scheme in groups with bilinear pairings [AFGHO16] to obtain a constant-sized commitment C to the matrix $\mathbf{M}$. This step imposes marginal overhead in the prover time and commitment key size. For simplicity of exposition, below we focus on Hyrax and defer the discussion of how to achieve RW streaming algorithms for Dory and BMMTV21 to Appendices B to D.

Opening. To prove that $p(\boldsymbol{z}) = v$, Open uses a 'vector-matrix-vector' product argument that allows a prover to convince a verifier that $\boldsymbol{\ell}^\top \mathbf{M} \boldsymbol{r} = v$ for a matrix $\mathbf{M}$ committed via the commitment $\boldsymbol{C}$, where $\boldsymbol{\ell}$ and $\boldsymbol{r}$ are public vectors constructed from $\boldsymbol{z}$ as above.

It is easy to see that $\boldsymbol{\ell}^\top \mathbf{M} = \sum_{i=0}^{\sqrt{N}-1} \ell_i \cdot \boldsymbol{M}_i$ and $\boldsymbol{\ell}^\top \mathbf{M} \boldsymbol{r} = \langle \sum_{i=0}^{\sqrt{N}-1} \ell_i \cdot \boldsymbol{M}_i, \boldsymbol{r} \rangle = v$. Thus, Hyrax's vector-matrix-vector product argument proceeds by having the verifier directly compute the commitment C to the vector $\boldsymbol{\ell}^\top \mathbf{M}$ by computing the linear combination of the row commitments with the $\boldsymbol{\ell}$ vector, $C := \sum_{i=0}^{\sqrt{N}-1} \ell_i \cdot C_i$. Open then uses an IPA to produce a proof that the inner product of the resulting committed vector C with $\boldsymbol{r}$ equals the claimed evaluation v.

Since vector-matrix multiplication can be computed in time $O(N)$ for matrices of dimension $\sqrt{N} \times \sqrt{N}$, and running an IPA over vectors of length $\sqrt{N}$ only takes $O(\sqrt{N})$ cryptographic time, Open requires $O(N)$ (*non-cryptographic*) time overall.

Barrier to read-only streaming. It seems that we cannot simultaneously achieve (cryptographic) linear-time and logarithmic random-access space for both the Commit and Open algorithms of VMV PC schemes. In particular, given access to the matrix $\mathbf{M}$ in row-major order, the Commit algorithm can be implemented in a read-only streaming manner with only logarithmic random-access memory. On the other hand, Open computes the vector-matrix product $\boldsymbol{\ell}^\top \mathbf{M}$, and streaming computation of the latter requires a *column-major-order* stream of $\mathbf{M}$. Since the surrounding application (in this case the SNARK) only provides row-major access to $\mathbf{M}$, Open would need to compute the transpose of $\mathbf{M}$ to obtain column-major access to $\mathbf{M}$, and even the best *in-place* (i.e. non-streaming) algorithms for this task require $O(N)$ time and space.

[17] The $\otimes$ operation denotes the Kronecker product. Given vectors $\boldsymbol{x} \in \mathbb{F}^N$ and $\boldsymbol{y} \in \mathbb{F}^M$, $\boldsymbol{x} \otimes \boldsymbol{y} := [x_1 \cdot \boldsymbol{y}, x_2 \cdot \boldsymbol{y}, \ldots, x_N \cdot \boldsymbol{y}]$.

Achieving RW streaming in cryptographic linear-time. We now outline how to construct a RW streaming algorithm for Open that achieves linear-time and logarithmic random-access space by leveraging just *two* intermediate write-streams of size $\sqrt{N}$. Our algorithm avoids the need for matrix transposition entirely.

Given streaming access to $\boldsymbol{\ell}$ and to the rows of the matrix $\mathbf{M}$ (denoted by $\boldsymbol{M}_0, \ldots, \boldsymbol{M}_{\sqrt{N}-1}$), the algorithm starts by reading ℓ_0, streaming through $\boldsymbol{M}_0$, and computing and writing out $\ell_0 \cdot \boldsymbol{M}_0$ to an intermediate read-write stream $\mathbf{W}$. In the next iteration, it reads ℓ_1 and streams through both the next row $\boldsymbol{M}_1$ *and* $\mathbf{W}$, and updates $\mathbf{W}$ to contain $\ell_0 \cdot \boldsymbol{M}_0 + \ell_1 \cdot \boldsymbol{M}_1$ by adding in the product $\ell_1 \cdot \boldsymbol{M}_1$.[18] This process continues until $\mathbf{W}$ contains the desired output $\ell_0 \cdot \boldsymbol{M}_0 + \ldots + \ell_{\sqrt{N}-1} \cdot \boldsymbol{M}_{\sqrt{N}-1}$. Clearly, throughout this process, the algorithm only consumes $O(\log N)$ random access space and $O(\sqrt{N})$ streaming space (for $\mathbf{W}$). We describe the full construction of Hyrax in Appendix C.3.

2.9 Implementation

We implemented SCRIBE in Rust atop the `arkworks` framework [con22] by adapting and extending the HyperPlonk implementation.[19] Our implementation is modular and allows for switching out almost all components, from the PC scheme to the underlying elliptic curve. We also adapt the `jellyfish` circuit construction framework[20] to output witness streams for the circuits it constructs. We provide next details about the infrastructure we developed to enable efficient and ergonomic implementation of read-write streaming algorithms.

2.9.1 Tools for read-write streams

To implement the read-write streaming algorithms outlined in Sections 2.6 and 2.8, we developed a slew of tools that allow for efficient streaming operations on vectors stored on disk. We believe that these tools will be of independent interest for future work on read-write streaming algorithms.

Low-overhead serialization and deserialization. To ensure that data can be efficiently streamed to and from disk, we implement custom serialization and deserialization routines for common types (e.g., integers, fields, and group elements). Unlike the standard serialization routines in `arkworks`, our routines are optimized for temporary storage and avoid expensive canonicalization steps (e.g., converting elliptic curve points between projective and affine coordinates).

File-backed vectors. We provide a new `FileVec` type whose API resembles that of Rust's standard `Vec` type, but which uses temporary files on disk as backing storage for the vector. We carefully engineer `FileVec` to ensure that its data can be accessed only in a streaming manner, and furthermore augment it to automatically switch to memory-backed storage when the vector becomes small enough. To ensure that accesses to `FileVec` do not populate the operating system's file-system cache, `FileVec` opens files with the `O_DIRECT` flag on Linux and the `F_NOCACHE` flag on MacOS.

File-backed multilinear polynomials. We leverage `FileVec` to obtain a multilinear polynomial type whose evaluations are stored on disk. We augment this type to avoid disk storage entirely whenever the polynomial

[18] To be more precise, because our model does not allow simultaneous read-write access to the same stream, the algorithm would need to use two intermediate read-write streams $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$, and alternate between them in each iteration to obtain the desired sum: once the algorithm has written out $\ell_0 \cdot \boldsymbol{M}_0$ onto $\boldsymbol{W}_1$, it must stream through both the running sum in $\boldsymbol{W}_1$ as well as $\boldsymbol{M}_1$ and compute and write out the new running sum $\ell_0 \cdot \boldsymbol{M}_0 + \ell_1 \cdot \boldsymbol{M}_1$ onto $\boldsymbol{W}_2$. It would then read this new running sum from $\boldsymbol{W}_2$ as well as ℓ_2 and $\boldsymbol{M}_2$ to write out $\ell_0 \cdot \boldsymbol{M}_0 + \ell_1 \cdot \boldsymbol{M}_1 + \ell_2 \cdot \boldsymbol{M}_2$ onto $\boldsymbol{W}_1$, and so on.

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

[20] https://github.com/EspressoSystems/jellyfish

can be streamed in a read-only manner (e.g., as is the case for the multilinear Lagrange polynomial eq.); this optimization greatly improved performance in our implementation.

Batched iterators. We implement the various stream operations required by our read-write streaming algorithms via a new *batched iterator* interface. We specify this interface as a Rust trait, `BatchedIterator`, whose API resembles Rust's standard `Iterator` trait, but which processes (in parallel via the `rayon` library[21]) a *batch* of elements on each iteration. `BatchedIterator` supports common operations such as `map`, `filter`, `enumerate`, and `zip`, and different `BatchedIterator` instances can be composed into complex pipelines that minimize disk I/O. We also implement `BatchedIterator` for `FileVec`.

2.10 Evaluation

We conduct a thorough evaluation of SCRIBE's performance in a variety of settings. Our evaluation aims to answer the following questions:

- **Q1:** Can SCRIBE scale to prove large circuits, and what overhead does it incur compared to state-of-the-art baselines (both low-memory and memory-intensive ones)?
- **Q2:** When does I/O bandwidth become a bottleneck?
- **Q3:** What is the memory cost of witness synthesis?

Baselines. We compare SCRIBE against two main baselines: the memory-intensive HyperPlonk [CBBZ23] and the low-memory Gemini [BCHO22].

Parameters. We configure all proof systems to use the BLS12-381 elliptic curve. SCRIBE's `FileVecs` switch over to memory-backed storage when they contain at most 2^{16} elements.

Main experimental setup. We perform our benchmarks on an AWS EC2 `im4gn.4xlarge` instance with an Intel Xeon Platinum 8175 CPU with 12 cores at 3.1 GHz, 7.5 TiB of storage, and 96 GiB of memory. The on-demand price of this instance is $1.356 per hour at the time of writing. To enforce memory and bandwidth limits, we rely on the `cgroups` functionality of Linux (via the `systemd-run` command). Both SCRIBE and Gemini are configured to use at most 2 GiB of memory, while HyperPlonk is allowed to use all available memory.

2.10.1 Scalability

We evaluate the performance of SCRIBE, Gemini, and HyperPlonk on our main experimental setup. All provers use 8 threads. Our results are shown in Fig. 3.

Running time. SCRIBE's latency scales linearly with instance size, and is $9.2\times$ smaller than Gemini's latency while being just $1.1\times$-$1.3\times$ larger than HyperPlonk's latency.

Scaling to large instances. Both Gemini and SCRIBE can scale to large instances, while HyperPlonk cannot prove circuits larger than 2^{24} gates due to memory constraints. We note that SCRIBE's superior running time allowed us to benchmark it on larger instances (up to 2^{28} gates) than Gemini (up to 2^{26} gates). While not shown in Fig. 3, we also evaluated industrial-quality SNARK libraries Plonky2 and Halo2 in the same setup, and found that they too could not scale to large instances in a limited-memory setting.

Performance on mobile devices. We also ran the same benchmarks on an iPhone 13 Pro Max to evaluate the performance of SCRIBE on a mobile device. Our experiments show that the overhead of SCRIBE compared

[21] https://github.com/rayon-rs/rayon

to HyperPlonk is similar to the main setup; however, SCRIBE can scale to $32\times$ larger instances. We note though that the advantage of SCRIBE over Gemini is reduced to just $6.5\times$.

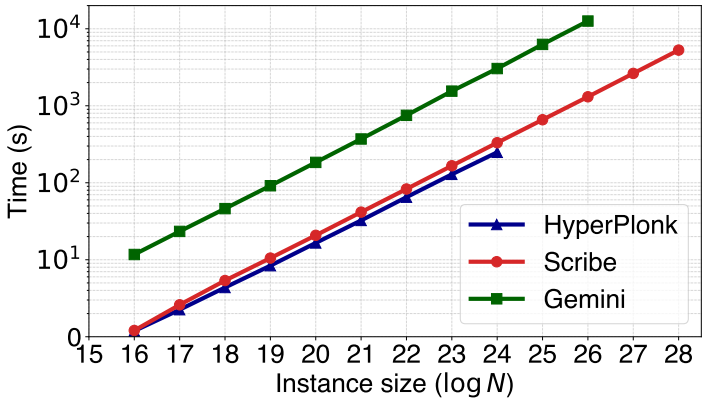

Figure 3: Proving time of SCRIBE, HyperPlonk, and Gemini.

2.10.2 Overhead of read-write-streaming

We investigate the overhead incurred by SCRIBE due to read-write streaming via two experiments, both of which simulate different compute-to-I/O ratios. The first varies the number of threads used by both SCRIBE and HyperPlonk and is reported in Fig. 4. Our results show that SCRIBE's overhead does not vary significantly with the number of threads, which indicates that for typical SSDs and for a reasonable number of threads, SCRIBE is not I/O bound.

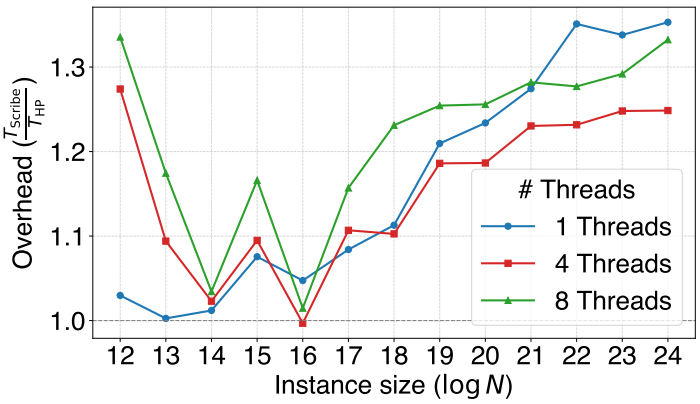

Figure 4: Overhead of SCRIBE over HyperPlonk as the number of threads varies.

Our second experiment fixes the number of threads to 8 and instead varies disk bandwidth. We report our results in Fig. 5, which shows that when bandwidth is significantly throttled, SCRIBE's latency degrades, indicating an I/O bottleneck. However, even this degraded latency is at worst $3\times$ that of HyperPlonk.

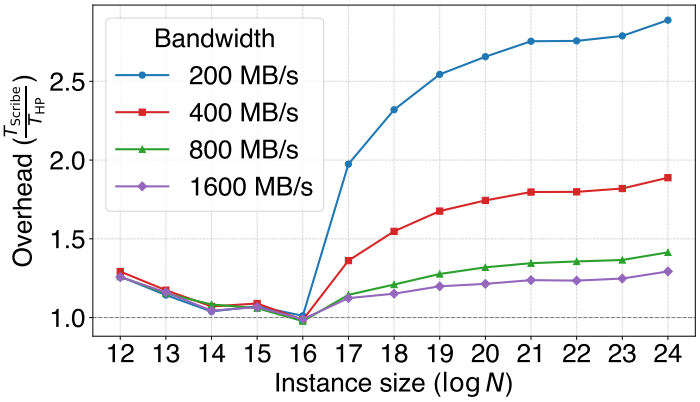

Figure 5: Overhead of SCRIBE over HyperPlonk when bandwidth is limited.

2.10.3 Cost of witness synthesis

To evaluate the time and memory costs of witness synthesis in our circuit programming framework, we sample randomly-generated circuits and try to synthesize witnesses for these. Our circuit-sampling process takes as input a number of gates, a size for a "working set" S that specifies the number of variables that are "in use" at any given moment during circuit generation, and a replacement probability that specifies the probability that a variable in the working set is replaced by a new variable. It produces a circuit C whose every gate is randomly sampled to be either an addition or a multiplication gate, and each gate's inputs are randomly sampled from the working set.

We evaluated the space and time costs of witness synthesis for circuits with sizes ranging from 2^{15} to 2^{28} with a variety of working set sizes. Our results are summarized in Table 3, which shows that these costs are a miniscule fraction of the corresponding proof generation costs even for all circuit sizes we evaluated.

working set size	memory	% of proving time
2^{15}	$< 68\,\text{MB}$	$< 1\%$
2^{17}	$< 72\,\text{MB}$	$< 1\%$
2^{20}	$< 110\,\text{MB}$	$< 1\%$

Table 3: Cost of witness synthesis for randomly-generated circuits C with $|C| \in \{\max(|S|, 2^{15}), \ldots, 2^{28}\}$.

3 Related work

3.1 Similar memory models

Read-write streaming. Prior work [GKS05; GS05; GHS06; BJR07; DFR09; FJM14] has considered a model of read-write streaming algorithms similar to ours, but focused on primitives such as sorting, graph algorithms, and database algorithms. The focus of these works is primarily theoretical, and they do not try to optimize for the running time of the algorithms, but rather only for the random-access space complexity and the number of passes over the external memory. Papakonstantinou and Yang [PY14] consider the possibility of implementing simple cryptographic primitives (like encryption) in a limited version of the read-write streaming model, and demonstrate some theoretical feasibility results.

External memory model. The external memory model [AV88] has been widely studied in the context of algorithms for large datasets that do not fit in RAM. The key parameter that is optimized in this model is the number of I/Os, which is the number of times the algorithm accesses the external memory (such as disk). Our RW streaming model is more restrictive than the external memory model because in the former, algorithms only have streaming access to external disk, and are also optimize for total work done instead of just the number of I/Os. Therefore, lower bounds in the external memory model also apply in our RW streaming model.

3.2 Read-only streaming SNARKs

A recent line of work attempts to tackle the prover space bottleneck by constructing 'streaming' SNARKs whose prover requires only a small amount of random-access space, and can only access the circuit wire values in a *read-only* streaming manner. This model differs from ours in that the prover does not have access to intermediate read-write streams. A number of prior works [BHRRS20; BHRRS21; BCHO22; ZCLKZ24] construct streaming SNARKs in this restricted model, but do so by sacrificing on prover time efficiency: the prover's algorithm in all these works incurs at least an $\Omega(\log N)$ factor overhead compared to non-streaming equivalents, and concrete efficiency is at least $10\times$ worse than state-of-the-art non-streaming SNARKs [Set20; CBBZ23]. We begin with a general comparison against the entire class.

Comparison of frameworks. The ability to use intermediate read-write streams enables more efficient algorithm composition than in the read-only streaming model, which in turn means that our generic construction of read-write streaming SNARKs in Corollary 2.4 achieve better concrete efficiency compared to the corresponding read-only streaming SNARKs. In more detail, the SNARK prover must feed the polynomials output by the PIOP prover to both the PC commitment algorithm and the PC opening algorithm. In the read-write setting, the SNARK prover can write the polynomials to intermediate read-write streams and use these as the input stream for both PC algorithms, but in the read-only setting the SNARK prover can neither store these polynomials in random-access memory nor write them out to disk, and so must compute them from scratch for each PC algorithm.

Block et al. [BHRRS20; BHRRS21] construct read-only streaming SNARKs by designing a streaming PIOP and combining it with streaming PC schemes from inner-product arguments [BHRRS20] or groups of unknown order [BHRRS21]. Both works achieve logarithmic random-access space complexity for the prover, but incur quasi-linear time complexity. They do not implement their SNARK constructions, which prevents us from comparing their concrete efficiency with ours. Our read-write streaming inner-product argument takes inspiration from the read-only streaming one in [BHRRS20], but, unlike the latter, is able to achieve cryptographic linear time complexity.

Gemini [BCHO22] generalizes the notion of streaming SNARKs and introduces *elastic* SNARKs that have prover algorithms optimized for two different memory regimes. The first regime is a memory-intensive one where the focus is on minimizing prover time, while the second regime is the read-only streaming setting where the focus is on minimizing prover memory. The latter regime is the relevant point of comparison for SCRIBE. We provide a quantitative comparison with Gemini's streaming prover in Section 2.10, and so focus here on a qualitative comparison. On an instance of size N, Gemini's streaming prover achieves $O(N \log^2 N)$ prover time with logarithmic prover memory, and does so by applying their analogue of the 'PIOP + PC $\rightarrow$ SNARK' transformation [CHMMVW20; BFS20] to streaming PIOPs and streaming PC schemes that they construct. Attempting to use read-write streams to improve the efficiency of their SNARK does not seem to work, as their PIOP requires multiplying a random vector by a sparse matrix, and this seems to inherently require quasi-linear time in a low-memory setting.

Epistle [ZCLKZ24] designs a new SNARK that improves upon Gemini: its streaming prover requires only $O(N \log N)$ time without increasing prover memory. Like SCRIBE, Epistle's starting point is the (memory-intensive) HyperPlonk SNARK [CBBZ23], but it switches out HyperPlonk's prodcheck PIOP in favor of a novel construction that is more amenable to read-only streaming. Like HyperPlonk's prodcheck, their new prodcheck PIOP also reduces to a sumcheck, but the key difference is that this reduction requires only $O(N)$ time even when restricted to $O(\log N)$ space. The key time complexity bottleneck of Epistle's PIOP is the sumcheck protocol, since it requires $O(N \log N)$ time in the read-only streaming model. Since Epistle's code is not publicly available at the time of writing, we are unable to provide a quantitative comparison of Epistle with SCRIBE. However, the numbers in their paper indicate that they incur an order-of-magnitude overhead over HyperPlonk, while our overhead is just 10-35%.

Blendy [CFFZ24] introduces a read-only streaming sumcheck protocol for multilinear polynomials which requires $O(N^{1/k})$ space and $O(kN)$ time, where k is a tunable parameter. Using $k = \log N / \log \log N$ this gives us a $O(\log N)$ space and $O(N \log N / \log \log N)$ time sumcheck protocol. Unfortunately, this work is limited to multilinear polynomials, which is insufficient for constructing SNARKs (which require sumcheck over products of multilinear polynomials).

Sparrow [PP24] is a read-only streaming SNARK whose prover requires $O(\sqrt{N})$ random-access space and $O(N \log \log N)$ time. Sparrow only supports data-parallel circuits. On a technical level, Sparrow introduces a new sumcheck protocol for products of multilinear polynomials (different from the standard one), and designs a read-only streaming PIOP for it that achieves the above efficiency.

3.3 Complexity-preserving SNARKs

Complexity-preserving SNARKs [BC12] impose at most a poly-logarithmic overhead in prover time and space compared to the corresponding costs for the computation being proven. We recap some relevant recent works in this space.

Low-memory SNARKs via IVC. Incrementally-verifiable computation (IVC) is the most popular means of constructing a complexity-preserving SNARK [BCCT12]. Unfortunately, IVC-based approaches have a number of drawbacks, including support for only uniform computations, non-black-box use of cryptography, a dependence on complex primitives like proof-carrying data [CT10] to achieve provable security, and, for many efficient constructions, a reliance on heuristics due to the need to instantiate random oracles. Nevertheless, IVC-based complexity-preserving SNARKs achieve good concrete efficiency. We discuss a few recent works.

Mangrove [NDCTB24] builds a complexity-preserving SNARK by reducing circuit-satisfiability to a uniform computation, and applying efficient accumulation/folding-based IVC schemes [BCLMS21; KST22] to this uniform computation. Mangrove's prover can be seen as a read-only streaming prover that makes two passes

over the witness, and requires $O(\log N)$ random-access space. Mangrove does not provide an implementation to compare against, and moreover suffers from many of the aforementioned limitations of IVC-based SNARKs.

Ligetron [WHV24] Ligetron builds an argument of knowledge with prover time $O(N \log N)$ and $O(\sqrt{N})$ random-access space. Unlike SCRIBE, Ligetron does not have a succinct verifier. Ligetron improves on the prior theoretical work of Bangalore et al. [BBHV22]. Unlike SCRIBE, both these works achieve provable security by relying on only the random oracle model (and no other cryptographic assumptions). We were unable to provide a quantitative comparison with Ligetron since their code is not publicly available at the time of writing.

4 Read-write streaming algorithms

In this section we define read-write (RW) streams in detail in Definition 4.1, and then describe the behavior of RW streaming algorithms in Definition 4.2. Subsequently, we describe common RW streaming algorithms in Section 4.1 that will be useful subroutines when designing RW streaming PIOPs and PC schemes in later sections.

Definition 4.1 (read-write streams)**.** *A* read-write stream $\mathbf{S}$ *is a tuple* $(\boldsymbol{a}, p, m, \ell)$ *where* $\boldsymbol{a}$ *is an underlying ordered set,* p *is a pointer to the ordered set, and* $m \in \{\mathsf{r}, \mathsf{w}\}$ *specifies whether* $\mathbf{S}$ *is in read mode (i.e. when* $m = \mathsf{r}$*) or write mode (i.e. when* $m = \mathsf{w}$*), and* ℓ *is the length of the underlying ordered set. Read-write streams support the following operations:*

- $\mathbf{S}.\mathsf{read}()$*: if* $\mathbf{S}.m = \mathsf{r}$*,* $\mathsf{read}()$ *returns the element at the current position of* $\mathbf{S}.p$*, and moves* $\mathbf{S}.p$ *to the next element of* $\mathbf{S}.\boldsymbol{a}$*.*
- $\mathbf{S}.\mathsf{write}(w)$*: if* $\mathbf{S}.m = \mathsf{w}$*,* $\mathsf{write}(w)$ *writes* w *at the current position of* $\mathbf{S}.p$ *(overwriting any existing value), and moves* $\mathbf{S}.p$ *to the next element of* $\mathbf{S}.\boldsymbol{a}$*.*
- $\mathbf{S}.\mathsf{init}(N)$ *initializes* $\mathbf{S} = (\boldsymbol{a}, p, \mathsf{w}, N)$ *in write mode by allocating a contiguous space for an array of* N *elements* $\boldsymbol{a}$*, and initializes* $\mathbf{S}.p$ *at the beginning of the allocated space. Note that the space allocated does not need to be initialized with any default values.*
- $\mathbf{S}.\mathsf{restart}()$ *resets* $\mathbf{S}.p$ *to the beginning of* $\mathbf{S}.\boldsymbol{a}$*.*
- $\mathbf{S}.\mathsf{swapmode}()$ *resets* $\mathbf{S}.p$ *to the beginning of* $\mathbf{S}.\boldsymbol{a}$ *and toggles* $\mathbf{S}.m$ *between* r *and* w*.*
- $\mathbf{S}.\mathsf{len}()$ *returns the length of the underlying set* $\mathbf{S}.\ell$*.*

Elements of underlying set. In the RW streaming algorithms described in subsequent sections, the underlying ordered set $\boldsymbol{a}$ of a RW stream $\mathbf{S}$ always contains elements from a field $\mathbb{F}$ or group $\mathbb{G}$, or constant-sized tuples thereof.

RW stream notation. We will typically denote a RW stream that only uses read mode as $\mathbf{R}$, a RW stream that only uses the write mode as $\mathbf{W}$, and a RW stream that uses both modes as $\mathbf{S}$. Additionally, we use $\mathbf{R}(\boldsymbol{a})$ to denote a RW stream in read mode where the underlying ordered set is $\boldsymbol{a}$ and the underlying pointer $\mathbf{S}.p$ is initialized to the beginning of $\boldsymbol{a}$.

Definition 4.2. *A read-write (RW) streaming algorithm* $\mathcal{A}(\mathbf{I}) \mapsto \mathbf{O}$ *is an algorithm that*

- *takes as input an input stream* $\mathbf{I}$ *and an output stream* $\mathbf{O}$ *to which* $\mathcal{A}$ *writes its output.*
- *can allocate a polylogarithmic number of intermediate streams using* init *and read/write intermediate states to them in a streaming manner using* $\mathsf{read}()$*,* $\mathsf{restart}()$*, and* $\mathsf{write}(\cdot)$*.*
- *can allocate at most polylogarithmic amount of random-access space that it can read and write to.*

Allocation of output streams. While the model dictates that $\mathcal{A}(\mathbf{I}) \mapsto \mathbf{O}$ does not allocate memory for $\mathbf{O}$, our pseudocode in subsequent sections will occasionally let $\mathcal{A}$ initialize $\mathbf{O}$ for clarity.

Input streams are immutable. Given RW streaming algorithm $\mathcal{A}(\mathbf{I}) \mapsto \mathbf{O}$, the memory that $\mathbf{I}$ points to can be considered immutable without loss of generality: given an input stream $\mathbf{I}$, $\mathcal{A}$ can allocate an RW stream $\mathbf{S}$ and then copy over the contents of $\mathbf{I}$ to $\mathbf{S}$ with one pass. Then $\mathcal{A}$ changes $\mathbf{S}$ to read mode with $\mathsf{swapmode}()$ and proceeds as if mutating the 'input read stream' $\mathbf{I}$ is allowed. In the following sections, input streams are occasionally treated as mutable to simplify pseudocode.

Single input and output streams. Restricting the number of input streams to 1 does not result in any loss of generality or asymptotic efficiency of the model; multiple input streams can be combined into a single stream

by concatenation, and the algorithm can then separate them out into different intermediate RW streams. In our pseudocode, we do allow multiple input streams as this improves concrete efficiency and makes the exposition simpler.

Space complexity. Consider a RW streaming algorithm $\mathcal{A}$. We define the *random-access space complexity* of $\mathcal{A}$ as the amount of random-access space allocated by $\mathcal{A}$. We define the *streaming space complexity* of $\mathcal{A}$ as $\sum_{i=1}^{k} \ell_i$ where ℓ_i is the length of the i-th intermediate stream allocated by $\mathcal{A}$. Both these complexities specifically count the amount of space *allocated* by $\mathcal{A}$, and not the total amount of space *used*. The distinction between the two is important because the former does not count the space required to store the input or output of $\mathcal{A}$ to avoid over-counting the space used when *composing* algorithms.

Definition 4.3 (composition of RW streaming algorithms)**.** *Consider 2 read-write streaming algorithms $\mathcal{C}(\mathbf{I}_1) \mapsto \mathbf{O}_1$ and $\mathcal{B}(\mathbf{I}_2) \mapsto \mathbf{O}_2$. We say that $\mathcal{A}(\mathbf{I}_1) \mapsto \mathbf{O}_1$ is the* composition *of $\mathcal{C}$ and $\mathcal{B}$ if it: (a) computes $\mathbf{I}_2$ inline using $\mathbf{I}_1$, (b) allocates space for $\mathbf{O}_2$, (c) calls $\mathcal{B}(\mathbf{I}_2) \mapsto \mathbf{O}_2$, (d) once $\mathcal{B}$ terminates, calls $\mathbf{O}_2$.swapmode(), (e) reads from $\mathbf{O}_2$ to compute its output, (f) writes output to $\mathbf{O}_1$.*

That is, $\mathcal{A}$ prepares the streaming input for $\mathcal{B}$, and then $\mathcal{A}$ uses the streaming output created by $\mathcal{B}$ to continue its computation. Informally, $\mathcal{A}$ calls $\mathcal{B}$ as a subroutine.

Lemma 4.4 (complexity of RW streaming algorithm composition)**.** *Let $\mathcal{A}, \mathcal{B}$ be read-write streaming algorithms. For algorithm $\mathcal{X} \in \{\mathcal{A}, \mathcal{B}\}$ let the time complexity be $t_{\mathcal{X}}$, random-access space complexity be $m_{\mathcal{X}}$, and streaming space complexity be $m_{\mathcal{X}}$. Let L be the number of distinct inputs that $\mathcal{A}$ passes to $\mathcal{B}$ over all its passes. Then $\mathcal{A}$ composed with $\mathcal{B}$ has*
- *time complexity $t_{\mathcal{A}} + L \cdot t_{\mathcal{B}}$,*
- *random-access space complexity $m_{\mathcal{A}} + m_{\mathcal{B}}$, and*
- *streaming space complexity $s_{\mathcal{A}} + s_{\mathcal{B}}$.*

Proof. Let the i-th unique call to $\mathcal{B}$ be $\mathcal{B}(\mathbf{R}_i) \mapsto \mathbf{W}_i$. $\mathcal{A}$ prepares each $\mathbf{R}_i$ and allocates space for each $\mathbf{W}_i$. The time and space required for these operations is already included in $t_{\mathcal{A}}$, $m_{\mathcal{A}}$, and $s_{\mathcal{A}}$. Each call to $\mathcal{B}$ takes $t_{\mathcal{B}}$ time, which implies a total additional time of $L \cdot t_{\mathcal{B}}$. However, each call to $\mathcal{B}$ also requires $m_{\mathcal{B}}$ random-access space and $s_{\mathcal{B}}$ streaming space, but this space can be reclaimed after each call to $\mathcal{B}$. Therefore the additional random-access space and streaming space required are only $m_{\mathcal{B}}$ and $s_{\mathcal{B}}$ respectively. □

Remark 4.5 (multiple subroutines)**.** Our notion of composition only allows an RW streaming algorithm to call at most one RW streaming algorithm as a subroutine. This is without loss of generality, as this ability is sufficient to simulate calls to multiple subroutines without much time or space overhead. For example, suppose $\mathcal{A}(\mathbf{R}) \mapsto \mathbf{W}$ calls two subroutines $\mathcal{B}_1(\mathbf{R}_1) \mapsto \mathbf{W}_1$ and $\mathcal{B}_2(\mathbf{R}_2) \mapsto \mathbf{W}_2$. We can rewrite $\mathcal{A}$ to instead call a single subroutine $\mathcal{B}'((\mathsf{flag}, \mathbf{R}')) \mapsto \mathbf{W}'$ defined as follows. The code for $\mathcal{B}'$ consists of a conditional statement that either executes $\mathcal{B}_1$'s code (which has been inlined into $\mathcal{B}'$), or makes a subroutine call to $\mathcal{B}_2$. In the input stream of $\mathcal{B}'$, flag indicates whether $\mathcal{B}_1$ should be executed or if a call to $\mathcal{B}_2$ should be made, and $\mathbf{R}'$ is the input to these algorithms (i.e., $\mathbf{R}' = \mathbf{R}_{\mathsf{flag}}$). It writes its output to $\mathbf{W}' = \mathbf{W}_{\mathsf{flag}}$, which is then read by $\mathcal{A}$. $\mathcal{A}$ can be rewritten to call $\mathcal{B}'$ with input $(1, \mathbf{R}_1)$ or $(2, \mathbf{R}_2)$ depending on whether $\mathcal{B}'$ needs to invoke $\mathcal{B}_1$ or $\mathcal{B}_2$. Clearly, this alternative uses the same amount of time and space as the original composition.

Remark 4.6 (Skipping allocation of intermediate streams)**.** Say RW streaming algorithm $\mathcal{A}(\mathbf{I}) \mapsto \mathbf{O}$ invokes $\mathcal{B}(\mathbf{I}) \mapsto \mathbf{S}$, and then invokes $\mathcal{C}(\mathbf{S}) \mapsto \mathbf{O}$. The model requires $\mathcal{A}$ to allocate $\mathbf{S}$, but without losing any asymptotic efficiency, $\mathcal{A}$ can skip this allocation and directly use the output of $\mathcal{B}$ as the input to $\mathcal{C}$, similar to the composition of read-only streaming algorithms. In fact, this leads to better concrete efficiency since it

avoids the overhead of allocating and initializing $\mathbf{S}$. We use the following notation to denote this optimization: if $\mathcal{B}$ outputs a stream $\mathbf{S}$ such that $\mathcal{A}$ does not need to allocate a RW stream for $\mathbf{S}$, then we write

$$\mathbf{S} \leftsquigarrow \mathcal{B}(\mathbf{I}) \ .$$

Furthermore, we extend this notation to compose multiple read-only streaming operations. If $\mathcal{A}$ can be implemented in a read-only manner using read-only implementations of $\mathcal{B}$ and $\mathcal{C}$, then we write $\mathcal{A}$ as

$$\mathcal{A} := \mathbf{O} \leftsquigarrow \mathcal{C} \leftsquigarrow \mathcal{B} \leftsquigarrow \mathbf{I} \ .$$

4.1 Common read-write streaming subroutines

We define below some helper functions that we use in our streaming algorithms in the rest of the paper.

Zip, map, and reduce. We introduce the Zip, Map, and Reduce subroutines that will be used in the RW streaming algorithms in the rest of the paper. Zip takes as input k streams $\mathbf{R}_1, \ldots, \mathbf{R}_k$ of length N containing k ordered sets, and outputs a stream $\mathbf{W}$ containing the zipped ordered set. That is, each element of the output stream is a tuple containing the elements of the input streams in the corresponding position. Map takes as input a stream $\mathbf{R}$ of length N and a function f, and outputs a stream $\mathbf{W}$ which contains the result of applying f to each element of $\mathbf{R}$. Reduce takes as input a stream $\mathbf{R}$ of length N, an identity element e, and an associative binary function f such that $f(e, b) = f(b, e) = b$ for any element of b in $\mathbf{R}$. The subroutine outputs a single element s that is the result of reducing the elements of $\mathbf{R}$ using f.

```
Zip(R_1, ..., R_k) ↦ W:
1. For i in [1, ..., R_1.len()]:
2.     W.write((R_1.read(),
3.         ..., R_k.read())).
```

```
Map(R, f) ↦ W:
1. For i in [1, ..., R.len()]:
2.     W.write(f(R.read())).
```

```
Reduce(R, f, e) ↦ s:
1. s ← e.
2. For i in [1, ..., R.len()]:
3.     s ← f(s, R.read()).
```

Splitting read-write streams. Given a RW stream $\mathbf{R}$ of length $N = 2^n$, the indices of the underlying ordered set can be represented as n-bit binary strings. That is, the i-th element of the ordered set is represented by $\mathsf{bin}_n(i)$. Given an additional parameter k, we can define two methods for splitting $\mathbf{R}$ into 2^k streams, each with $N/2^k$ elements. SplitMSB splits $\mathbf{R}$ on the k most significant bits of the indices, while SplitLSB splits $\mathbf{R}$ on the k least significant bits of the indices.

```
SplitMSB(R, k) ↦ (W_1, ..., W_{2^k}):
1. For j in [1, ..., 2^k]:
2.     For i in [1, ..., R.len()/2^k]: W_j.write(R.read()).
```

```
SplitLSB(R, k) ↦ (W_1, ..., W_{2^k}):
1. For i in [1, ..., R.len()/2^k]:
2.     For j in [1, ..., 2^k]: W_j.write(R.read()).
```

We additionally define $\mathsf{SplitLR}(\mathbf{R}) \mapsto (\mathbf{W}_L, \mathbf{W}_R)$ and $\mathsf{SplitEO}(\mathbf{R}) \mapsto (\mathbf{W}_E, \mathbf{W}_O)$ that are specific cases of SplitMSB and SplitLSB respectively, where the output consists of exactly two streams. That is, SplitLR splits $\mathbf{R}$ into a stream of the first half of the elements of $\mathbf{R}$ and a stream of the second half of the elements, while SplitEO splits $\mathbf{R}$ into a stream of the elements with even indices and a stream of the elements with odd indices. We omit the pseudocode for these.

Joining read-write streams. We also define the "inverse" algorithms JoinLR and JoinEO that take as input two RW streams and respectively concatenate them or interleave them; we omit the pseudocode for these.

kSum and kFoldInPlace. kSum takes as input a parameter d, and k RW streams which represent the evaluations of k many n-variate multilinear polynomials $p_1(\boldsymbol{X}), \ldots, p_k(\boldsymbol{X})$ on $\{0,1\}^n$. Consider the

univariate k-degree polynomial $f(X_n) = \sum_{\boldsymbol{b}\in\{0,1\}^{n-1}} \prod_{i=1}^{k} p_i(\boldsymbol{b}, X_n)$. kSum outputs $d+1$ evaluations $f(0), f(1), \ldots, f(d)$ of f.

kFoldInPlace also takes as input k RW streams which represent the evaluations of k many n-variate multilinear polynomials $p_1(\boldsymbol{X}), \ldots, p_k(\boldsymbol{X})$ on $\{0,1\}^n$. The method folds each polynomial p_i into a $(n-1)$-variate multilinear polynomial q_i into a new stream of half the length based on the folding coefficients α and β. That is, $q_i(\boldsymbol{X}) = p_i(\boldsymbol{X}, 0) \cdot \alpha + p_i(\boldsymbol{X}, 1) \cdot \beta$. Note that the space needed to store input streams of kFoldInPlace is freed at the end of the subroutine since the folded streams are moved into the input RW streams. This subroutine is used repeatedly in sumcheck-like protocols (such as in the polynomial commitment schemes described in Appendix C.3), where the output streams are the input streams for the next call to kFoldInPlace, and there is no need to store the intermediate folded streams.

kSum$(d, \mathbf{R}_1, \ldots, \mathbf{R}_k) \mapsto S$:
1. Set $S \leftarrow [0]_{s=0}^{d}$.
2. For i in $[1, \ldots, \mathbf{R}.\mathsf{len}()/2]$:
3. For j in $[1, \ldots, k]$:
4. $a_{L,j} \leftarrow \mathbf{R}_j.\mathsf{read}(); a_{R,j} \leftarrow \mathbf{R}_i.\mathsf{read}()$.
5. For s in $[0, \ldots, d]$:
6. $S[s] \leftarrow S[s] + \prod_{j=1}^{k} ((1-s) \cdot a_{L,j} + s \cdot a_{R,j})$.

kFoldInPlace$(\alpha, \beta, \mathbf{R}_1, \ldots, \mathbf{R}_k)$:
1. Define the folding function: $f(a, b) := a \cdot \alpha + b \cdot \beta$.
2. For i in $[1, \ldots, k]$:
3. Set $\mathbf{W}_i \leftarrow \mathsf{Map}(f) \leftsquigarrow \mathsf{Zip} \leftsquigarrow \mathsf{SplitEO} \leftsquigarrow \mathbf{R}_i$.
4. Set $\mathbf{R}_i \leftarrow \mathbf{W}_i.\mathsf{swapmode}()$.

Inner products and linear combinations of streams. Given RW streams $\mathbf{R}_1$ and $\mathbf{R}_2$, we describe RW streaming algorithms to compute the inner product of the underlying vectors, and to compute a linear combination with respect to coefficients α and β:

InnerProd$(\mathbf{R}_1, \mathbf{R}_2) \mapsto s$:
1. $s \leftsquigarrow \mathsf{Reduce}(0, +) \leftsquigarrow \mathsf{Map}(\times) \leftsquigarrow \mathsf{Zip} \leftsquigarrow (\mathbf{R}_1, \mathbf{R}_2)$.

LinComb$(\alpha, \beta, \mathbf{R}_1, \mathbf{R}_2) \mapsto \mathbf{W}$:
1. Define the function $f(a, b) := \alpha \cdot a + \beta \cdot b$.
2. Output $\mathbf{W} \leftsquigarrow \mathsf{Map}(f) \leftsquigarrow \mathsf{Zip} \leftsquigarrow (\mathbf{R}_1, \mathbf{R}_2)$.

5 Read-write streaming PIOP for HyperPlonk

In this section we construct a read-write streaming PIOP for the HyperPlonk relation. We will follow the following roadmap to achieve this goal: (a) formally define indexed relations and PIOPs in Section 5.1, (b) develop read-write streaming PIOPs for different instances of the sumcheck relation in Section 5.2, (c) develop read-write streaming PIOPs for the zerocheck relation (Section 5.3.1) and the permcheck realtion (Section 5.3.4), which in turn leads to the construction of a read-write streaming PIOP for the HyperPlonk relation in Section 5.3.5.

5.1 Preliminaries

5.1.1 Indexed relations

An *indexed relation* $\mathscr{R}$ is a set of tuples $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ where $\mathbb{i}$ is the index, $\mathbb{x}$ is the instance, and $\mathbb{w}$ is the witness. We use $\mathscr{R}_{\mathbb{i}}$ to denote the NP relation $\{(\mathbb{x}, \mathbb{w}) : (\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathscr{R}\}$ and $\mathscr{L}_{\mathbb{i}}$ to denote the NP language corresponding to it. An indexed *oracle* relation is an indexed relation where the index $\mathbb{i}$ and the instance $\mathbb{x}$ contain 'implicit' inputs that are specified as oracles, i.e., the membership-checking algorithm for such a relation has only query access to these oracles. We adopt notation from Chen et al. [CBBZ23] and use $[\![z]\!]$ to denote when the input z is provided as an oracle.

5.1.2 Polynomial IOPs

A **Polynomial Interactive Oracle Proof** (PIOP) over a field family $\mathcal{F}$ for an indexed relation $\mathscr{R}$ is a tuple $\mathsf{PIOP} = (\mathsf{k}, \mathsf{s}, \mathsf{I}, \mathsf{P}, \mathsf{V})$ where $\mathsf{k}, \mathsf{s} \colon \{0,1\}^* \to \mathbb{N}$ are polynomial-time functions and $\mathsf{I}, \mathsf{P}, \mathsf{V}$ are three algorithms known as the *indexer*, *prover*, and *verifier*. The parameter k specifies the number of rounds of interaction between the prover and the verifier, and s specifies the number of polynomials in each round.

In the offline phase, before the instance and witness are specified, the indexer I receives as input a field $\mathbb{F} \in \mathcal{F}$ and an index $\mathbb{i}$ for $\mathscr{R}$, and outputs $\mathsf{s}(0)$ polynomials $p_{0,1}, \ldots, p_{0,\mathsf{s}(0)}$.

In the online phase, given an instance $\mathbb{x}$ and witness $\mathbb{w}$ such that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathscr{R}$, the prover P receives $(\mathbb{F}, \mathbb{i}, \mathbb{x}, \mathbb{w})$ and the verifier V receives $(\mathbb{F}, \mathbb{x})$ and oracle access to the polynomials output by $\mathsf{I}(\mathbb{F}, \mathbb{i})$. The prover P and the verifier V interact over $\mathsf{k} = \mathsf{k}(|\mathbb{i}|)$ rounds.

For $i \in [\mathsf{k}]$, in the i-th round of interaction, the verifier V sends a message $\mu_i \in \mathbb{F}^*$ to the prover P; then the prover P replies with $\mathsf{s}(i)$ oracle polynomials $p_{i,1}, \ldots, p_{i,\mathsf{s}(i)}$. The verifier may query any of the polynomials it has received any number of times. A query consists of a location $z \in \mathbb{F}$ for an oracle $p_{i,j}$, and its corresponding answer is $p_{i,j}(z) \in \mathbb{F}$. After the interaction, the verifier accepts or rejects. Every PIOP must satisfy the following properties.

Completeness. For every field $\mathbb{F} \in \mathcal{F}$ and index-instance-witness tuple $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathscr{R}$, the probability that $\mathsf{P}(\mathbb{F}, \mathbb{i}, \mathbb{x}, \mathbb{w})$ convinces $\mathsf{V}^{\mathsf{I}(\mathbb{F},\mathbb{i})}(\mathbb{F}, \mathbb{x})$ to accept in the interactive oracle protocol is 1.

Knowledge soundness. We say that PIOP has knowledge error ϵ if there exists a probabilistic polynomial-time extractor E such that for every field $\mathbb{F} \in \mathcal{F}$, index $\mathbb{i}$, instance $\mathbb{x}$, and malicious prover $\tilde{\mathsf{P}}$,

$$\Pr\left[\begin{array}{c} (\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathscr{R} \\ \wedge \\ \langle \tilde{\mathsf{P}}, \mathsf{V}^{\mathsf{I}(\mathbb{F},\mathbb{i})}(\mathbb{F}, \mathbb{x}) \rangle = 1 \end{array} \middle| \; \mathbb{w} \leftarrow \mathsf{E}^{\tilde{\mathsf{P}}}(\mathbb{F}, \mathbb{i}, \mathbb{x}) \right] = \epsilon$$

The *query complexity* q is the total number of queries made by the verifier to the indexer and prover polynomials.

Additional notation and assumptions. In this paper, we only consider public-coin PIOPs where the verifier makes non-adaptive queries. We consider n-variate polynomials over a finite field $\mathbb{F}$ and assume that $N = 2^n$, and that N is much smaller than $|\mathbb{F}|$. RW streaming provers have access to an input polynomial p as a RW stream of evaluations on $\{0,1\}^n$ in a lexicographic order $[p(\mathsf{bin}(0)), p(\mathsf{bin}(1)), \ldots, p(\mathsf{bin}(N-1))]$, and we denote this stream as $\mathbf{R}(p)$. We omit the verifier's code for brevity, since they are unmodified from the non-streaming versions.

5.2 RW streaming prover for sumcheck

Extending the techniques discussed for multilinear polynomials in Section 2.5, we build a RW streaming sumcheck prover for a class of multivariate polynomials that we call *sumprod* polynomials (see Definition 5.1) in Section 5.2.1. In Section 5.2.2, we then present the Lagrange sumcheck: a special case of the sumprod sumcheck where one of the constituent polynomials is the Lagrange polynomial. Then, in Section 5.2.3, we build a RW streaming prover for *batch* sumcheck, which allows proving sumcheck claims about multiple sumprod polynomials simultaneously. These are key building blocks for the RW streaming algorithms that follow in Section 5.3.

5.2.1 Sumcheck for sumprod polynomials

We now describe our read-write streaming prover for the sumcheck relation for *sumprod* polynomials, which are multivariate polynomials that can be expressed as sums of products of multilinear polynomials. We define this class next, and show how to generalize the sumcheck prover for multilinear polynomials (Section 2.5.1) to this class.

Definition 5.1. *The class $\mathcal{P}_n^{\leq d} \subset \mathbb{F}^{\leq d}[X_1, \ldots, X_n]$ consists of n-variate polynomials of individual degree at most d that can be represented as a sum of products of d multilinear polynomials, for some $d \in \mathbb{N}$. That is, a polynomial $p(\boldsymbol{X})$ is in $\mathcal{P}_n^{\leq d}$ if there exist d multilinear polynomials $p_1(\boldsymbol{X}), p_2(\boldsymbol{X}), \ldots, p_d(\boldsymbol{X})$ and a multilinear polynomial h such that $p(\boldsymbol{X}) = h(p_1(\boldsymbol{X}), p_2(\boldsymbol{X}), \ldots, p_d(\boldsymbol{X}))$.*

Remark 5.2. The assumption that h is multilinear in Definition 5.1 is without loss of generality. We can "linearize" a k-variate polynomial h' where each variable has degree at most d into a (kd)-variate multilinear polynomial h where each d-degree variable of h' is replaced by d linear variables in h.

Definition 5.3. *The relation $\mathscr{R}_{\mathsf{SSC}}$ contains tuples of the form*

$$(\mathbb{i}_{\mathsf{SSC}}, \mathbb{x}_{\mathsf{SSC}}, \mathbb{w}_{\mathsf{SSC}}) = ((\mathbb{F}, n, d, h), ([\![p_1]\!], \ldots, [\![p_d]\!], \sigma), (p_1, \ldots, p_d))$$

where $\sigma \in \mathbb{F}$ is the target sum, each p_i is an n-variate multilinear polynomial, and h is a multilinear polynomial with ℓ monomials such that $\sum_{\boldsymbol{x} \in \{0,1\}^n} h(p_1(\boldsymbol{x}), \ldots, p_d(\boldsymbol{x})) = \sigma$.

We obtain a RW streaming prover for this relation by modifying the algorithm in Section 2.5.1 as follows. We first consider a single product of d multilinear polynomials. Recall that in the multilinear case, the round polynomials a_i sent by P in each round are linear, and therefore 2 evaluations are sufficient to specify them. Therefore, P computes 2 evaluations of the a_i's and sends them to V. For a product of d multilinear polynomials, each polynomial a_i is a univariate polynomial of degree d, and P specifies this via $d+1$ evaluations, which it sends to V.

To generalize to the sumprod polynomial $h(p_1, \ldots, p_d)$, P still computes $d+1$ evaluations of each round polynomial, but now it does so by iterating through all monomials in h, and computing and summing

evaluations for each of these. That is, for each monomial, P computes $d+1$ evaluations of the respective round polynomial (regardless of the degree of the monomial), and adds them to a running sum. At the end of the round, P is left with $d+1$ running sums corresponding to $d+1$ round polynomial evaluations, which it sends to V.

Read-write Streaming Algorithm 1: SUMCHECK for sumprod polynomials

$\mathsf{P}(\mathbb{i}_{\mathsf{SSC}}, \mathbb{x}_{\mathsf{SSC}}, (\mathbf{R}_1(p_1), \ldots, \mathbf{R}_d(p_d)))$:

1. For each i in $[n, \ldots, 1]$:
2. Initialize running sums: $S \leftarrow [0]_{j=0}^{d}$.
3. For each monomial $(c \cdot X_{j_1} \cdot X_{j_2} \cdot \ldots \cdot X_{j_k})$ in h:
4. Obtain $d+1$ evaluations for the round polynomial of the monomial: set $E \leftarrow \mathsf{kSum}(d, \mathbf{R}_{j_1}, \ldots, \mathbf{R}_{j_k})$.
5. Add each evaluation to the corresponding running sum: for s in $[0, \ldots, d]$: set $S[s] \leftarrow S[s] + c \cdot E[s]$.
6. Restart the streams that were used for this monomial: $\mathbf{R}_{j_1}.\mathsf{restart}(), \ldots, \mathbf{R}_{j_k}.\mathsf{restart}()$.
7. Send $[a_i(s) := S[s]]_{s=0}^{d}$ to V.
8. If $i \neq 1$:
9. Receive verifier challenge $r_i \xleftarrow{\$} \mathbb{F}$ from V.
10. Fold each stream individually: $\mathsf{kFoldInPlace}(1 - r_i, r_i, \mathbf{R}_1, \ldots, \mathbf{R}_d)$.

We prove the following lemma which helps us show that the prover sends the correct values to the verifier in Step 7 for any iteration of the loop in Step 1.

Lemma 5.4. *Let q be an input polynomial of Algorithm 1 such that $\mathbf{S}$ contains its evaluations over $\{0,1\}^n$. Also, let q_i be the i-variate polynomial such that $\mathbf{S}$ contains its evaluations over $\{0,1\}^i$ at the beginning of iteration i of Step 1. Then $q_i(\mathsf{bin}(j)) = q(\mathsf{bin}(j), r_{i+1}, \ldots, r_n)$ for all $j \in \{0, \ldots, N_i - 1\}$.*

Proof. We proceed by induction and consider the first iteration of Step 1 when $i = n$. At the beginning of the iteration, $q_n(\mathsf{bin}(j)) = q(\mathsf{bin}(j))$ for all $j \in \{0, \ldots, N-1\}$ as desired.

Now consider $i < n$. By the induction hypothesis, we have $q_{i+1}(\mathsf{bin}(j)) = q(\mathsf{bin}(j), r_{i+2}, \ldots, r_n)$ for all $j \in \{0, \ldots, N_{i+1} - 1\}$ at the beginning of iteration $i+1$. At the end of the iteration $i+1$ (and before the start of iteration i), Step 10 creates a stream of $N_i/2$ elements such that

$$\begin{aligned} q_i(\mathsf{bin}(j)) &= (1 - r_{i+1}) \cdot q_{i+1}(\mathsf{bin}(2j), r_{i+2}, \ldots, r_n) + r_{i+1} \cdot q_{i+1}(\mathsf{bin}(2j+1), r_{i+2}, \ldots, r_n) \\ &= (1 - r_{i+1}) \cdot q(\mathsf{bin}(2j), r_{i+2}, \ldots, r_n) + r_{i+1} \cdot q(\mathsf{bin}(2j+1), r_{i+2}, \ldots, r_n) \\ &= (1 - r_{i+1}) \cdot q(\mathsf{bin}(j), 0, r_{i+2}, \ldots, r_n) + r_{i+1} \cdot q(\mathsf{bin}(j), 1, r_{i+2}, \ldots, r_n) \\ &= q(\mathsf{bin}(j), r_{i+1}, \ldots, r_n) \end{aligned}$$

for all $j \in \{0, \ldots, N_{i-1} - 1\}$ as desired. □

Lemma 5.5. *PIOP 1 and Algorithm 1 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{SSC}}$ for n-variate (d, ℓ)-sumprod polynomials with the following efficiency properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(d\ell N)$	$O(d+\ell)$	$O(dN)$	$O(d)$	$O(d \log N/\lvert\mathbb{F}\rvert)$	$O(d \log N)$	$O(d)$

Proof. We will prove the statement for the product of 2 polynomials p and q that received in two input streams $\mathbf{R}_p$ and $\mathbf{R}_q$. The proof extends to (d, ℓ)-sumprod polynomials straightforwardly. Completeness and soundness proofs follow from [Tha22] if we can show that the messages sent by P in Step 7 are correct. That is, we need to show that $S[\alpha] = a_i(\alpha)$ for any $\alpha \in \mathbb{F}$.

If p_i is the i-variate polynomial that $\mathbf{R}_p$ contains at the beginning of iteration i of Step 1 in Algorithm 1, then $p_i(\mathsf{bin}(j)) = p(\mathsf{bin}(j), r_{i+1}, \ldots, r_n)$ for all $j \in \{0, \ldots, N_i - 1\}$ due to Lemma 5.4 (similarly $q_i(\mathsf{bin}(j)) = q(\mathsf{bin}(j), r_{i+1}, \ldots, r_n)$). We have

$$\begin{aligned} a_i(\alpha) &= \sum_{\boldsymbol{b} \in \{0,1\}^{i-1}} p(\boldsymbol{b}, \alpha, r_{i+1}, \ldots, r_n) \cdot q(\boldsymbol{b}, \alpha, r_{i+1}, \ldots, r_n) \\ &= \sum_{j=0}^{N_i/2-1} ((1-\alpha)p_i(\mathsf{bin}(2j)) + \alpha p_i(\mathsf{bin}(2j+1))) \cdot ((1-\alpha)q_i(\mathsf{bin}(2j)) + \alpha q_i(\mathsf{bin}(2j+1))) = S[\alpha] \end{aligned}$$

as desired.

P requires $O(d\ell N_i)$ time in the ith round of the protocol, and therefore requires $O(d\ell N)$ time in total. Moreover, P only requires $O(d)$ random-access space to store the round polynomial evaluations, and $O(\ell)$ random-access space to store h. Finally, P requires $O(dN)$ streaming space across d distinct streams to store the folded multilinear polynomials. V makes one query for each multilinear polynomial and therefore makes d queries in total, and the communication is $O(d \log N)$ since P sends $d + 1$ evaluations of $\log N$ round polynomials. □

5.2.2 Lagrange sumcheck for sumprod polynomials

We additionally propose a RW streaming prover for the sumprod sumcheck relation where one of the constituent polynomials is the (partially-evaluated) multilinear Lagrange polynomial (i.e., $\mathrm{eq}(\boldsymbol{t}, \boldsymbol{X})$ for some $\boldsymbol{t}$).

Definition 5.6. *The relation $\mathscr{R}_{\mathsf{LSC}}$ contains tuples of the form*

$$(\mathbb{i}_{\mathsf{LSC}}, \mathbb{x}_{\mathsf{LSC}}, \mathbb{w}_{\mathsf{LSC}}) = ((\mathbb{F}, n, d, h), (\llbracket p_1 \rrbracket, \ldots, \llbracket p_d \rrbracket, \boldsymbol{t}, \sigma), (p_1, \ldots, p_d))$$

where $\sigma \in \mathbb{F}$ is the target sum, each p_i is an n-variate multilinear polynomial, $\boldsymbol{t} \in \mathbb{F}^n$, and h is a multilinear polynomial with ℓ monomials such that $\sum_{\boldsymbol{x} \in \{0,1\}^n} h(p_1(\boldsymbol{x}), \ldots, p_d(\boldsymbol{x})) \cdot \mathrm{eq}(\boldsymbol{t}, \boldsymbol{x}) = \sigma$.

Since this relation is a special case of the sumprod sumcheck relation Definition 5.3, we use the same PIOP as in PIOP 1, and describe an efficient RW streaming prover specifically for this relation. We define a helper method kSumEq, which is similar to the kSum method defined in Section 4.1, but additionally takes as input a vector $\boldsymbol{t} \in \mathbb{F}^n$, which describes the Lagrange polynomial.

Read-write Streaming Algorithm 2: LAGRANGE SUMCHECK for sumprod polynomials

$\mathsf{P}(\mathbb{i}_{\mathsf{LSC}}, \mathbb{x}_{\mathsf{LSC}}, (\mathbf{R}_1(p_1), \ldots, \mathbf{R}_d(p_d)))$:

1. Initialize folding coefficient for $\mathrm{eq}(\boldsymbol{t}, \boldsymbol{X})$: $\gamma \leftarrow 1$.
2. For each i in $[n, \ldots, 1]$:
3. Initialize running sums: $S \leftarrow [0]_{j=0}^{d+1}$.
4. For each monomial $(c \cdot X_{j_1} \cdot X_{j_2} \cdot \ldots \cdot X_{j_k})$ in h:
5. Get $d+2$ evaluations for the monomial's round polynomial: set $E \leftarrow \mathsf{kSumEq}(d+1, i, \boldsymbol{t}, \mathbf{R}_{j_1}, \ldots, \mathbf{R}_{j_k})$.
6. Add each evaluation to the corresponding running sum: for s in $[0, \ldots, d]$: set $S[s] \leftarrow S[s] + c \cdot E[s]$.
7. Restart the streams that were used for this monomial: $\mathbf{R}_{j_1}.\mathsf{restart}(), \ldots, \mathbf{R}_{j_k}.\mathsf{restart}()$.
8. Send $[a_i(s) := \gamma \cdot S[s]]_{s=0}^{d+1}$ to V.
9. If $i \neq 1$:
10. Receive verifier challenge $r_i \xleftarrow{\$} \mathbb{F}$ from V.
11. Fold each stream individually: $\mathsf{kFoldInPlace}(1 - r_i, r_i, \mathbf{R}_1, \ldots, \mathbf{R}_d)$.
12. Obtain folded Lagrange polynomial by updating folding coefficient: $\gamma \leftarrow \gamma \cdot (t_i \cdot r_i + (1 - t_i) \cdot (1 - r_i))$.

$\mathsf{kSumEq}(d, i, \boldsymbol{t}, \mathbf{R}_1, \ldots, \mathbf{R}_k) \mapsto S$:

1. $\mathsf{Eq.Init}(t_1, t_2, \ldots, t_i)$.
2. Set $S \leftarrow [0]_{s=0}^{d}$.
3. For i in $[1, \ldots, \mathbf{R}.\mathsf{len}()/2]$:
4. For j in $[1, \ldots, k]$:
5. $a_{L,j} \leftarrow \mathbf{R}_j.\mathsf{read}(); a_{R,j} \leftarrow \mathbf{R}_i.\mathsf{read}()$.
6. $a_{L,k+1} \leftarrow \mathsf{Eq.Next}; a_{R,k+1} \leftarrow \mathsf{Eq.Next}$.
7. For s in $[0, \ldots, d]$:
8. $S[s] \leftarrow S[s] + \prod_{j=1}^{k+1} ((1 - s) \cdot a_{L,j} + s \cdot a_{R,j})$

Lemma 5.7. *PIOP 1 and Algorithm 2 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{LSC}}$ for n-variate (d, ℓ)-sumprod polynomials with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(d\ell N)$	$O(d + \ell + \log N)$	$O(dN)$	$O(d)$	$O(d \log N/\lvert\mathbb{F}\rvert)$	$O(d \log N)$	$O(d)$

Proof. All properties follow from Lemma 5.5 except for the random-access space complexity, which increases to $O(d + \ell + \log N)$ due to the additional space required to store $\boldsymbol{t}$. The time complexity does not increase because each call to kSumEq in iteration i of Step 2 still requires $O(d \cdot 2^i)$ time for each monomial in h, since the Lagrange polynomial evaluations can be streamed in $O(2^i)$ time.

We additionally need to prove that the messages sent by P in each iteration of Step 2 are correct. That is, we need to show that for all $\alpha \in \{0, \ldots, d+1\}$,

$$\gamma_i \cdot S[\alpha] = \sum_{\boldsymbol{x} \in \{0,1\}^{i-1}} h(p_1(\boldsymbol{x}, \alpha, r_{i+1}, \ldots, r_n), \ldots, p_d(\boldsymbol{x}, \alpha, r_{i+1}, \ldots, r_n)) \cdot \mathrm{eq}((\boldsymbol{x}, \alpha, r_{i+1}, \ldots, r_n), \boldsymbol{t}) \ ,$$

where γ_i is the folding coefficient at the beginning of the iteration i of Step 2. We prove the above statement for a single multilinear polynomial q for brevity, the proof extends to (d, ℓ)-sumprod polynomials straightforwardly, using the same arguments as in Lemma 5.5. Let q_i be the polynomial stored in the input stream $\mathbf{R}$ in iteration i of Step 2. We have

$$\sum_{\boldsymbol{x} \in \{0,1\}^{i-1}} q(\boldsymbol{x}, \alpha, r_{i+1}, \ldots, r_n) \cdot \mathrm{eq}(\boldsymbol{t}, (\boldsymbol{x}, \alpha, r_{i+1}, \ldots, r_n))$$

$$= \sum_{\boldsymbol{x}\in\{0,1\}^{i-1}} q_i(\boldsymbol{x},\alpha)\cdot \mathrm{eq}(\boldsymbol{t},(\boldsymbol{x},\alpha,r_{i+1},\ldots,r_n)) \qquad \text{[Lemma 5.4]}$$

$$= \prod_{j=i+1}^{n}((1-r_j)(1-t_j)+r_jt_j)\cdot \sum_{\boldsymbol{x}\in\{0,1\}^{i-1}} q_i(\boldsymbol{x},\alpha)\cdot \mathrm{eq}((t_1,\ldots,t_i),(\boldsymbol{x},\alpha))$$

$$= \gamma_i \cdot \sum_{\boldsymbol{x}\in\{0,1\}^{i-1}} q_i(\boldsymbol{x},\alpha)\cdot \mathrm{eq}((t_1,\ldots,t_i),(\boldsymbol{x},\alpha)) = \gamma_i\cdot S[\alpha]$$

as desired. □

5.2.3 Batch sumcheck for sumprod polynomials

We finally present the batch RW streaming sumcheck for multiple (d,ℓ)-sumprod polynomials. Let $p_1,\ldots,p_\nu$ be n-variate (d,ℓ)-sumprod polynomials, and let $\sigma_1,\ldots,\sigma_\nu$ be their claimed sums on $\{0,1\}^n$ respectively. We can check all these claims simultaneously via the standard random linear combination technique.

In more detail, P obtains a random challenge α from V, and runs a RW streaming sumcheck PIOP for the (d,ℓ)-sumprod polynomial $p(\boldsymbol{X}) = \sum_{i=1}^{\nu}\alpha^{i-1}p_i(\boldsymbol{X})$, and target sum $\sigma = \sum_{i=1}^{\nu}\alpha^{i-1}\sigma_i$. This transformation incurs a negligible soundness error.

We now define the batch sumcheck relation. Let $p_{(i,j)}$ be the jth constituent multilinear polynomial of p_i. That is, $p_i = h_i(p_{(i,1)},\ldots,p_{(i,d)})$. V has oracle access to each $p_{(i,j)}$ individually, and P has streaming access to all $d\nu$ of them individually.

$$\begin{aligned}\mathscr{R}_{\mathsf{BSC}} &= (\mathtt{i}_{\mathsf{BSC}},\mathtt{x}_{\mathsf{BSC}},\mathtt{w}_{\mathsf{BSC}})\\ &= \left((\mathbb{F},n,d,\nu,h_1,\ldots,h_\nu),\left([\![p_{(1,1)}]\!],\ldots,[\![p_{(\nu,d)}]\!],\sigma_1,\ldots,\sigma_\nu\right),(p_{(1,1)},\ldots,p_{(\nu,d)})\right)\end{aligned}$$

We present a RW streaming prover for $\mathscr{R}_{\mathsf{BSC}}$ whose efficiency is determined by the efficiency of the underlying sumcheck protocol for sumprod polynomials.

Read-write Streaming Algorithm 3: BATCH SUMCHECK for multiple (d,ℓ)-sumprod polynomials

$\mathsf{P}(\mathtt{i}_{\mathsf{BSC}},\mathtt{x}_{\mathsf{BSC}},(\mathbf{R}_{(1,1)}(p_{(1,1)}),\ldots,\mathbf{R}_{(\nu,d)}(p_{(\nu,d)}))):$

1. P receives $\alpha \xleftarrow{\$} \mathbb{F}$ from V.
2. Initialize batch target sum $\sigma \leftarrow \sum_{i=1}^{\nu}\alpha^{i-1}\sigma_i$.
3. Define batch polynomial $h(X_{1,1},\ldots,X_{1,d},\ldots,X_{\nu,1},\ldots,X_{\nu,d}) := \sum_{i=1}^{\nu}\alpha^{i-1}h_i(X_{i,1},\ldots,X_{i,d})$
4. Define $\mathtt{i} := (\mathbb{F},n,d\nu,h)$ and $\mathtt{x} := ([\![p_{(1,1)}]\!],\ldots,[\![p_{(\nu,d)}]\!],\sigma)$ and $\mathtt{w} := (\mathbf{R}_{(1,1)},\ldots,\mathbf{R}_{(\nu,d)})$
5. P and V invoke the RW streaming sumcheck PIOP for a $(d\nu,\ell\nu)$-sumprod polynomial for $(\mathtt{i},\mathtt{x},\mathtt{w})$ defined above.

5.3 Read-write streaming prover for HyperPlonk's PIOP

In this section we present RW streaming PIOPs for several relations including zerocheck (Section 5.3.1), prodcheck (Section 5.3.2), multiset-equality-check (Section 5.3.3), and permcheck (Section 5.3.4).

Then, in Section 5.3.5 we show how to use these PIOPs to create a RW streaming PIOP for the HyperPlonk relation [CBBZ23], which in turn directly gives us a RW streaming PIOP for circuit satisfiability as described in Section 2.2. For each PIOP, we first recall the standard non-streaming implementation of the prover, and then present a RW streaming version for the same.

5.3.1 Zerocheck PIOP

Given an n-variate polynomial p, the zerocheck PIOP checks if p is zero at all points of an n-dimensional boolean hypercube $\{0,1\}^n$. This is formalized via the following relation for (d,ℓ)- sumprod polynomials:

Definition 5.8. *The zerocheck relation $\mathscr{R}_{\mathsf{ZC}}$ for (d,ℓ)-sumprod polynomials is an indexed relation consisting of the following tuples:*

$$(\mathtt{i}_{\mathsf{ZC}}, \mathtt{x}_{\mathsf{ZC}}, \mathtt{w}_{\mathsf{ZC}}) = ((\mathbb{F}, n, d, h), (\llbracket p_1 \rrbracket, \ldots, \llbracket p_d \rrbracket), (p_1, \ldots, p_d))$$

where each p_i is an n-variate multilinear polynomial, and h is a multilinear polynomial with ℓ terms such that $h(p_1(\boldsymbol{x}), \ldots, p_d(\boldsymbol{x})) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$.

The PIOP below illustrates a standard way of proving this relation.

PIOP 3: ZEROCHECK

$\langle \mathsf{P}(\mathtt{i}_{\mathsf{ZC}}, \mathtt{x}_{\mathsf{ZC}}, \mathtt{w}_{\mathsf{ZC}}), \mathsf{V}(\mathtt{i}_{\mathsf{ZC}}, \mathtt{x}_{\mathsf{ZC}}) \rangle$:

1. P receives $\boldsymbol{r} \xleftarrow{\$} \mathbb{F}^n$ from V.
2. P computes the multilinear polynomial $q(\boldsymbol{X}) := \mathrm{eq}(\boldsymbol{X}, \boldsymbol{r})$.
3. P and V invoke the sumcheck PIOP for the claim "$\sum_{\boldsymbol{x} \in \{0,1\}^n} h(p_1(\boldsymbol{x}), \ldots, p_d(\boldsymbol{x})) \cdot q(\boldsymbol{x}) = 0$".

Prior work (for example, [CBBZ23]) provides completeness and soundness proof for this PIOP. We show how to construct a streaming prover for (d,ℓ)-sumprod polynomials using Algorithm 1. Let h be the multilinear polynomial such that $p(\boldsymbol{X}) = h(p_1(\boldsymbol{X}), \ldots, p_d(\boldsymbol{X}))$. P and V have streaming and oracle access to p_i respectively.

Read-write Streaming Algorithm 4: ZEROCHECK for (d,ℓ)-sumprod polynomials

$\mathsf{P}(\mathtt{i}_{\mathsf{ZC}}, (\llbracket p_1 \rrbracket, \ldots, \llbracket p_d \rrbracket), \mathtt{w}_{\mathsf{ZC}})$:

1. Receive $\boldsymbol{r} \xleftarrow{\$} \mathbb{F}^n$ from V.
2. Define Lagrange sumcheck instance: $\mathtt{x} := (\llbracket p_1 \rrbracket, \ldots, \llbracket p_d \rrbracket, \llbracket q \rrbracket, \boldsymbol{r}, 0)$
3. Invoke the RW streaming Lagrange sumcheck PIOP for sumprod polynomials for $(\mathtt{i}_{\mathsf{ZC}}, \mathtt{x}, \mathtt{w}_{\mathsf{ZC}})$.

Lemma 5.9. *PIOP 3 and Algorithm 4 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{ZC}}$ for n-variate (d,ℓ)-sumprod polynomials with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(d\ell N)$	$O(d+\ell+\log N)$	$O(dN)$	$O(d)$	$O(d \log N/\lvert\mathbb{F}\rvert)$	$O(d \log N)$	$O(d)$

Proof. The PIOP properties follow from the analysis of Chen et al. [CBBZ23] because the prover in Algorithm 4 sends the same values to V as in PIOP 3, which follows from Lemma 5.5.

An additional $\log N$ random-access space is required to store $\boldsymbol{r}$ while computing $\mathrm{eq}(\boldsymbol{X}, \boldsymbol{r})$. The total prover time required to compute $\mathrm{eq}(\boldsymbol{X}, \boldsymbol{r})$ is $O(N)$ due to the amortized efficiency of $\mathsf{Eq.Next}$ as discussed in Section 2.6.1. Other efficiency parameters follow from Lemma 5.5. □

Remark 5.10. Algorithm 4 can be viewed as an interactive reduction from zerocheck to sumcheck. This protocol will be used alongside other PIOPs that will also reduce to sumcheck, therefore it will be helpful to perform a single batch sumcheck for all the sumcheck claims. To do so, we extract a method ZToS that performs Step 1 to Step 2 of Algorithm 4, and emits the resulting sumcheck claim.

5.3.2 Prodcheck PIOP

Given n-variate polynomials p and q and a target product σ, the prodcheck PIOP tests if the product of $p(\boldsymbol{x})/q(\boldsymbol{x})$ is σ over all $\boldsymbol{x} \in \{0,1\}^n$. This is formalized via the following relation:

Definition 5.11 (Prodcheck relation)**.** *The prodcheck relation $\mathscr{R}_{\mathsf{PDC}}$ is an indexed relation consisting of tuples $(\mathbb{i}_{\mathsf{PDC}}, \mathbb{x}_{\mathsf{PDC}}, \mathbb{w}_{\mathsf{PDC}}) = ((\mathbb{F}, n), (\llbracket p \rrbracket, \llbracket q \rrbracket, \sigma), (p, q))$, where p, q are n-variate multilinear polynomials such that $\prod_{\boldsymbol{x} \in \{0,1\}^n} p(\boldsymbol{x})/q(\boldsymbol{x}) = \sigma$.*

The following PIOP (due to Setty and Lee [SL20]) illustrates a standard way of proving this relation.

PIOP 4: PRODCHECK

$\langle \mathsf{P}(\mathbb{i}_{\mathsf{PDC}}, \mathbb{x}_{\mathsf{PDC}}, \mathbb{w}_{\mathsf{PDC}}), \mathsf{V}(\mathbb{i}_{\mathsf{PDC}}, \mathbb{x}_{\mathsf{PDC}}) \rangle$:

1. P sends $(n+1)$-variate polynomial ν such that $\nu(0, \boldsymbol{X}) := p(\boldsymbol{X})/q(\boldsymbol{X})$ and $\nu(1, \boldsymbol{X}) := \nu(\boldsymbol{X}, 0) \cdot \nu(\boldsymbol{X}, 1)$.
2. Define $f(\boldsymbol{X}) := \nu(0, \boldsymbol{X}) \cdot q(\boldsymbol{X}) - p(\boldsymbol{X})$ and $g(\boldsymbol{X}) := \nu(1, \boldsymbol{X}) - \nu(\boldsymbol{X}, 0) \cdot \nu(\boldsymbol{X}, 1)$.
3. P and V invoke the zerocheck PIOP for $f(\boldsymbol{X})$ and $g(\boldsymbol{X})$.
4. V checks that $\nu(1, 1, \ldots, 1, 0) = \sigma$.

Prior work [SL20; CBBZ23] provides completeness and soundness proof for this PIOP. We now describe our read-write streaming prover for this PIOP. As discussed in Section 2.6.2, the computation of the polynomial ν induces a binary-tree structure. We define the $(n-i)$-variate polynomial $\nu_i(\boldsymbol{X}) = \nu(1, \ldots, 1, 0, \boldsymbol{X})$ where the first i variables of ν are fixed to 1, and the $(i+1)$-th variable is fixed to 0. P computes ν_i for $i = 0, \ldots, n$ in that order, since an evaluation of ν_{i+1} is the product of two evaluations of ν_i. Finally, it concatenates the $n+1$ streams to get one stream for ν.

Read-write Streaming Algorithm 5: PRODCHECK

$\mathsf{P}(\mathbb{i}_{\mathsf{PDC}}, \mathbb{x}_{\mathsf{PDC}}, (\mathbf{R}_p(p), \mathbf{R}_q(q)))$:

Initialize RW streams for ν_j as defined above:

1. Allocate space for each ν_j: for j in $[0, \ldots, n]$: $\mathbf{S}_{\nu,j}.\mathsf{init}(N/2^j)$.
2. Compute stream for polynomial $\nu_0(\boldsymbol{X}) := \nu(0, \boldsymbol{X})$: $\mathbf{S}_{\nu,0} \leftarrow \mathsf{Map}(/) \rightsquigarrow \mathsf{Zip} \rightsquigarrow (\mathbf{R}_p, \mathbf{R}_q)$.

Compute each ν_i using the binary tree structure:

3. For j in $[1, \ldots, n]$:
4. $\quad \mathbf{S}_{\nu,j-1}.\mathsf{swapmode}()$.
5. $\quad \mathbf{S}_{\nu,j} \leftarrow \mathsf{Map}(\times) \rightsquigarrow \mathsf{Zip} \rightsquigarrow \mathsf{SplitEO} \rightsquigarrow \mathbf{S}_{\nu,j-1}$.
6. $\mathbf{S}_\nu \leftarrow \mathsf{Concat}(\mathbf{S}_{\nu,0}, \mathbf{S}_{\nu,1}, \ldots, \mathbf{S}_{\nu,n}, [0])$ (append 0 at the end to make the stream length a power of 2).
7. Create $\llbracket \nu \rrbracket$ from $\mathbf{S}_\nu$ and send to V.

Split stream of ν into halves for the zerocheck instances:

8. Define $\nu_L(\boldsymbol{X}) := \nu(0, \boldsymbol{X})$ and $\nu_R(\boldsymbol{X}) := \nu(1, \boldsymbol{X})$ and $\nu_E(\boldsymbol{X}) := \nu(\boldsymbol{X}, 0)$ and $\nu_O(\boldsymbol{X}) := \nu(\boldsymbol{X}, 1)$.
9. Split ν to obtain streams for $\nu(\boldsymbol{X}, 0)$ and $\nu(\boldsymbol{X}, 1)$: $(\mathbf{R}_{\nu,E}, \mathbf{R}_{\nu,O}) \leftarrow \mathsf{SplitEO}(\mathbf{S}_\nu)$.
10. Split ν to obtain streams for $\nu(0, \boldsymbol{X})$ and $\nu(1, \boldsymbol{X})$: $(\mathbf{R}_{\nu,L}, \mathbf{R}_{\nu,R}) \leftarrow \mathsf{SplitLR}(\mathbf{S}_\nu)$.

Initialize sumcheck instances using ZToS:

11. Define $t_0(x_1, x_2, x_3) := x_1 \cdot x_2 - x_3$.
12. Obtain sumcheck claim for the zerocheck claim "$\nu_L(\boldsymbol{X}) \cdot q(\boldsymbol{X}) - p(\boldsymbol{X}) = 0$":
    $(\mathbb{i}_f = (\_, \_, \_, t_1), \mathbb{x}_f, \mathbb{w}_f) \leftarrow \mathsf{ZToS}((\mathbb{F}, n, 2, t_0), (\llbracket \nu_L \rrbracket, \llbracket q \rrbracket, \llbracket p \rrbracket), (\mathbf{R}_{\nu,L}, \mathbf{R}_q, \mathbf{R}_p))$.
13. Obtain sumcheck claim for the zerocheck claim "$\nu_E(\boldsymbol{X}) \cdot \nu_O(\boldsymbol{X}) - \nu_R(\boldsymbol{X}) = 0$":
    $(\mathbb{i}_g = (\_, \_, \_, t_2), \mathbb{x}_g, \mathbb{w}_g) \leftarrow \mathsf{ZToS}((\mathbb{F}, n, 2, t_0), (\llbracket \nu_E \rrbracket, \llbracket \nu_O \rrbracket, \llbracket \nu_R \rrbracket), (\mathbf{R}_{\nu,E}, \mathbf{R}_{\nu,O}, \mathbf{R}_{\nu,R}))$.
14. Define $\mathbb{i} := (\mathbb{F}, n, 3, 2, t_1, t_2)$ and $\mathbb{x} := (\mathbb{x}_f, \mathbb{x}_g)$ and $\mathbb{w} := (\mathbb{w}_f, \mathbb{w}_g)$.
15. Invoke the RW streaming batch sumcheck PIOP for sumprod polynomials for $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ defined above, which batches the two sumcheck claims into one.

Remark 5.12. In the above algorithm V implicitly has polynomial oracle access to $\nu_L, \nu_R, \nu_E, \nu_O$ due to having polynomial oracle access to ν.

Lemma 5.13. *PIOP 4 and Algorithm 5 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{PDC}}$ with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(N)$	$O(\log N)$	$O(N)$	$O(1)$	$O(N/\lvert\mathbb{F}\rvert)$	$O(\log N)$	$O(1)$

Proof. The PIOP properties follow from the analysis of Chen et al. [CBBZ23] because P in Algorithm 5 produces the same values as PIOP 4. The time and space efficiency parameters follow from Lemma 5.9 where $d = O(1)$. □

5.3.3 Multiset-equality-check PIOP

We use the read-write streaming PIOP for prodcheck to build a read-write streaming PIOP for proving claims about equality of multisets encoded as multilinear polynomials.

Definition 5.14 (multiset-equality)**.** *The indexed relation $\mathscr{R}_{\mathsf{MEC}}$ consists of tuples $(\mathbb{i}_{\mathsf{MEC}}, \mathbb{x}_{\mathsf{MEC}}, \mathbb{w}_{\mathsf{MEC}}) = ((\mathbb{F}, n), (\llbracket p \rrbracket, \llbracket q \rrbracket), (p, q))$ where p and q are n-variate multilinear polynomials such that the multisets $\{\!\{p(\boldsymbol{x})\}\!\}_{\boldsymbol{x}\in\{0,1\}^n}$ and $\{\!\{q(\boldsymbol{x})\}\!\}_{\boldsymbol{x}\in\{0,1\}^n}$ are equal.*

The PIOP below illustrates a way of proving this as described by Chen et al. [CBBZ23]:

PIOP 5: MULTISET-EQUALITY-CHECK

$\langle \mathsf{P}(\mathbb{i}_{\mathsf{MEC}}, \mathbb{x}_{\mathsf{MEC}}, \mathbb{w}_{\mathsf{MEC}}), \mathsf{V}(\mathbb{i}_{\mathsf{MEC}}, \mathbb{x}_{\mathsf{MEC}}) \rangle$:

1. P receives a random challenge $\alpha \in \mathbb{F}$ from V.
2. P computes polynomials $p'(\boldsymbol{X}) := \alpha + p(\boldsymbol{X})$ and $q'(\boldsymbol{X}) := \alpha + q(\boldsymbol{X})$.
3. P and V invoke the prodcheck PIOP for the claim "$\prod_{\boldsymbol{x}\in\{0,1\}^n} p'(\boldsymbol{x})/q'(\boldsymbol{x}) = 1$".

Below we demonstrate a streaming prover for this that avoids intermediate read-write streams.

Read-write Streaming Algorithm 6: MULTISET-EQUALITY-CHECK

$\mathsf{P}(\mathbb{i}_{\mathsf{MEC}}, \mathbb{x}_{\mathsf{MEC}}, (\mathbf{R}_p(p), \mathbf{R}_q(q)))$:

1. Receive a random challenge $\alpha \in \mathbb{F}$ from V.

   Initialize read-only streams for $p'(\boldsymbol{X}) := \alpha + p(\boldsymbol{X})$ and $q'(\boldsymbol{X}) := \alpha + q(\boldsymbol{X})$:
2. $\mathbf{S}_{p'} \rightsquigarrow \mathsf{Map}(x \mapsto \alpha + x) \rightsquigarrow \mathbf{R}_p$.
3. $\mathbf{S}_{q'} \rightsquigarrow \mathsf{Map}(x \mapsto \alpha + x) \rightsquigarrow \mathbf{R}_q$.

   Reduce to prodcheck:
4. Define $\mathbb{i} := (\mathbb{F}, n); \mathbb{x} := (\llbracket p' \rrbracket, \llbracket q' \rrbracket, 1); \mathbb{w} := (\mathbf{S}_{p'}, \mathbf{S}_{q'})$
5. Invoke the RW streaming prodcheck PIOP for $(\mathbb{i}, \mathbb{x}, \mathbb{w})$.

Lemma 5.15. *PIOP 5 and Algorithm 6 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{MEC}}$ with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(N)$	$O(\log N)$	$O(N)$	$O(1)$	$O(N/\lvert\mathbb{F}\rvert)$	$O(\log N)$	$O(1)$

Proof. PIOP properties follow from the analysis of Chen et al. [CBBZ23] because P in Algorithm 6 sends the same values to V as in PIOP 5. Moreover, Algorithm 6 has the same time and space complexity as the RW streaming prodcheck PIOP. □

5.3.4 Permcheck PIOP

We use the foregoing read-write streaming PIOP for multiset-equality to design a read-write streaming PIOP for proving that two lists are permutations.

Definition 5.16 (permcheck)**.** *The index relation $\mathscr{R}_{\mathsf{PC}}$ consists of tuples $(\mathbb{i}_{\mathsf{PC}}, \mathbb{x}_{\mathsf{PC}}, \mathbb{w}_{\mathsf{PC}}) = ((\mathbb{F}, n, \pi), (\llbracket p \rrbracket, \llbracket q \rrbracket, \llbracket \hat{\pi} \rrbracket), (p, q, \hat{\pi}))$ where p, q are multilinear polynomials and $\hat{\pi}$ is the multilinear extension of a permutation $\pi : [N] \to [N]$ such that such that $p(\boldsymbol{x}) = q(\mathsf{bin}(\pi(\mathsf{int}(\boldsymbol{x}))))$ for all $\boldsymbol{x} \in \{0,1\}^n$.*

The PIOP below illustrates a way of proving this as described by Chen et al. [CBBZ23]:

PIOP 6: PERMCHECK

$\langle \mathsf{P}(\mathbb{i}_{\mathsf{PC}}, \mathbb{x}_{\mathsf{PC}}, \mathbb{w}_{\mathsf{PC}}), \mathsf{V}(\mathbb{i}_{\mathsf{PC}}, \mathbb{x}_{\mathsf{PC}}) \rangle$:

1. P receives a random challenge $\beta \in \mathbb{F}$ from V.
2. P computes polynomials $p'(\boldsymbol{X}) := p(\boldsymbol{X}) + \hat{\pi}(\boldsymbol{X}) \cdot \beta$ and $q'(\boldsymbol{X}) := q(\boldsymbol{X}) + \mathsf{int}(\boldsymbol{X}) \cdot \beta$.
3. P and V invoke the multiset-equality PIOP for the claim "$\{\!\{p'(\boldsymbol{x}) : \boldsymbol{x} \in \{0,1\}^n\}\!\} = \{\!\{q'(\boldsymbol{x}) : \boldsymbol{x} \in \{0,1\}^n\}\!\}$".

Below we demonstrate a streaming prover for this PIOP that avoids intermediate read-write streams.

Read-write Streaming Algorithm 7: PERMCHECK

$\mathsf{P}(\mathbb{i}_{\mathsf{PC}}, \mathbb{x}_{\mathsf{PC}}, (\mathbf{R}_p(p), \mathbf{R}_q(q), \mathbf{R}_{\hat{\pi}}(\hat{\pi})))$:

1. Receive a random challenge $\beta \in \mathbb{F}$ from V.

   Initialize read-only streams for $p'(\boldsymbol{X}) := p(\boldsymbol{X}) + \hat{\pi}(\boldsymbol{X}) \cdot \beta$ and $q'(\boldsymbol{X}) := q(\boldsymbol{X}) + \mathsf{int}(\boldsymbol{X}) \cdot \beta$:
2. $\mathbf{S}_{p'} \rightsquigarrow \mathsf{Map}((x,y) \mapsto x + \beta \cdot y) \rightsquigarrow \mathsf{Zip} \rightsquigarrow (\mathbf{R}_p, \mathbf{R}_{\hat{\pi}})$.
3. $\mathbf{S}_{q'} \rightsquigarrow \mathsf{Map}((x,y) \mapsto x + \beta \cdot y) \rightsquigarrow \mathsf{Zip} \rightsquigarrow \big(\mathbf{R}_q, [i]_{i=1}^N\big)$.

   Reduce to multiset-equality-check:
4. Define $\mathbb{i} := (\mathbb{F}, n)$ and $\mathbb{x} := (\llbracket p \rrbracket, \llbracket q \rrbracket)$ and $\mathbb{w} := (\mathbf{S}_{p'}, \mathbf{S}_{q'})$
5. Invoke the RW streaming multiset-equality-check PIOP for $(\mathbb{i}, \mathbb{x}, \mathbb{w})$.

Lemma 5.17. *PIOP 6 and Algorithm 7 together comprise a read-write streaming PIOP for for $\mathscr{R}_{\mathsf{PC}}$ for n-variate multilinear polynomials with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(N)$	$O(\log N)$	$O(N)$	$O(1)$	$O(N/\lvert\mathbb{F}\rvert)$	$O(\log N)$	$O(1)$

Proof. PIOP properties follow from the analysis of Chen et al. [CBBZ23] because P in Algorithm 7 sends the same values to V as in PIOP 6. Moreover, Algorithm 7 has the same time and space complexity as the RW streaming multiset-equality-check PIOP.

□

Remark 5.18. Similar to the RW streaming prover for the zerocheck PIOP, Algorithm 7 can be viewed as an interactive reduction from permcheck to sumcheck. Similar to ZToS, we extract a method PToS that performs Step 1 to Step 4 of Algorithm 7, and emits the resulting sumcheck claim.

5.3.5 HyperPlonk PIOP

We now discuss HyperPlonk's circuit representation, and then define the HyperPlonk relation $\mathscr{R}_{\mathsf{HPC}}$.

Definition 5.19. *For an arithmetic circuit $C : \mathbb{F}^k \to \mathbb{F}$, the* HyperPlonk circuit representation *is a tuple $A(C) = \left(\mathbb{F}, d, N, N_p, N_w, N_q, q, \pi, f_0, \ldots, f_{N_q-1}\right)$ where*
- *$\mathbb{F}$ is a finite field,*
- *N is the number of gates in C,*
- *N_p is the number of public inputs to C,*
- *$N_w - 1$ is the arity of C,*
- *N_q is the number of different types of gates in C,*
- *$q : \{0, \ldots, N-1\} \to \{0, \ldots, N_q - 1\}$ is a selector function such that $q(i)$ is the "type" of the i-th gate,*
- *$\pi : \{0, \ldots, N_w - 1\} \times \{0, \ldots, N-1\} \to \{0, \ldots, N_w - 1\} \times \{0, \ldots, N-1\}$ describes the wiring identity constraints, and*
- *$f_j : \mathbb{F}^{N_w - 1} \to \mathbb{F}$ is a degree-d map that describes the behavior of the j-th type of gate.*

We assume that N, N_p, N_w, N_q are powers of 2, i.e., that there exist $n, n_p, n_w, n_q \in \mathbb{N}$ such that $N = 2^n$, $N_p = 2^{n_p}$, $N_w = 2^{n_w}$, and $N_q = 2^{n_q}$.

Definition 5.20. *The HyperPlonk relation $\mathscr{R}_{\mathsf{HPC}}$ is an indexed relation consisting of the following tuples:*

$$(\mathbb{i}_{\mathsf{HPC}}, \mathbb{x}_{\mathsf{HPC}}, \mathbb{w}_{\mathsf{HPC}}) = (A, (\llbracket \boldsymbol{w} \rrbracket, \boldsymbol{p}), (\boldsymbol{w}))$$

where $A = \left(\mathbb{F}, d, N, N_p, N_w, N_q, q, \pi, f_0, \ldots, f_{N_q-1}\right)$ is the arithmetic representation of C according to Definition 5.19, $\boldsymbol{p} \in \mathbb{F}^{N_p}$ is the public input vector, and $\boldsymbol{w} \in \mathbb{F}^{N_w \times N}$ is the witness vector where $w_{i,j}$ is the i-th input of gate j if $i < N_w - 1$ and the output of gate j if $i = N_w - 1$.
These satisfy the following constraints:
- *wiring identity: $w_{i,j} = w_{\pi(i,j)}$ for all $i \in \{0, \ldots, N_w - 1\}, j \in \{0, \ldots, N-1\}$.*
- *gate identity: $f_{q(i)}\left(w_{0,i}, \ldots, w_{N_w-2,i}\right) = w_{N_w-1,i}$ for all $i \in \{0, \ldots, N-1\}$.*
- *public input consistency: check if the initial part of $\boldsymbol{w}$ is consistent with the public input. That is, for all $j \in [0, \ldots, N_p - 1]$,*

$$w_{i,j} = \begin{cases} p_j : & i = N_w - 1 \\ 0 : & i < N_w - 1 \end{cases}.$$

Remark 5.21. In Definition 5.20, N_w, N_q, d are all small constants and only depend on the arity of the circuit, number of different types of gates in the circuit, and maximum degree of a gate. For example, for simple binary circuits with only addition and multiplication gates, $N_w = 3, N_q = 2, d = 1$. Therefore $f_0, \ldots, f_{N_q-1}$ have constant-sized descriptions that can be stored in memory by both P and V.

Multilinear polynomial representation. Since $\mathscr{R}_{\mathsf{HPC}}$ is defined in the context of *vectors* instead of polynomials, we need a few more definitions to describe the PIOP for $\mathscr{R}_{\mathsf{HPC}}$ so that we can reduce it to the aforementioned zerocheck and permcheck PIOPs:
- $\tilde{p} : \{0,1\}^{n_p} \to \mathbb{F}$ where $\tilde{p}(\mathsf{bin}_{n_p}(i)) = p_i$.
- $\tilde{w} : \{0,1\}^{n_w+n} \to \mathbb{F}$ where $\tilde{w}(\mathsf{bin}_{n_w}(i), \mathsf{bin}_n(j)) = w_{i,j}$.
- $\tilde{q} : \{0,1\}^{n_q+n} \to \mathbb{F}$ where

$$\tilde{q}(\mathsf{bin}_{n_q}(i), \mathsf{bin}_n(j)) = \begin{cases} 1 : & q(j) = i \\ 0 : & \text{otherwise} \end{cases}.$$

- $\tilde{\pi} : \{0,1\}^{n_w+n} \to \mathbb{F}$ where $\tilde{\pi}(\mathsf{bin}_{n_w}(i), \mathsf{bin}_n(j)) = a + N_w \cdot b.$ such that $(a, b) = \pi(i, j)$.
- Define the "combining gate function" as follows:

$$\tilde{f}(\boldsymbol{X}, Y_0, Y_1, \ldots, Y_{N_w-1}) = f_{t(\boldsymbol{X})}(Y_0, Y_1, \ldots, Y_{N_w-2}) - Y_{N_w-1}$$

  where $\boldsymbol{X} \in \{0,1\}^{N_q}$ and $t(\boldsymbol{X})$ is the smallest index such that $X_t = 1$.

We additionally define $\hat{p}, \hat{w}, \hat{q}, \hat{\pi}$ as the multilinear extensions of $\tilde{p}, \tilde{w}, \tilde{q}, \tilde{\pi}$ respectively. We also define $\hat{f} : \mathbb{F}^n \to \mathbb{F}$ as follows:

$$\hat{f}(\boldsymbol{X}) = \tilde{f}\left(\hat{q}(\mathsf{bin}_{n_q}(0), \boldsymbol{X}), \ldots, \hat{q}(\mathsf{bin}_{n_q}(N_q - 1), \boldsymbol{X}), \hat{w}(\mathsf{bin}_{n_w}(0), \boldsymbol{X}), \ldots, \hat{w}(\mathsf{bin}_{n_w}(N_w - 1), \boldsymbol{X})\right)$$

By definition of $\hat{f}$, only one of the first N_q inputs to $\tilde{f}$ is equal to 1, and the rest are 0; if the $t(\boldsymbol{X})$-th input is 1, then clearly the $\boldsymbol{X}$-th gate of the circuit has type $t(\boldsymbol{X})$. Therefore, if the gate constraint is satisfied, then we must have $\hat{f}(\boldsymbol{X}) = f_{t(\boldsymbol{X})}(\hat{w}(\mathsf{bin}(0), \boldsymbol{X}), \ldots, \hat{w}(\mathsf{bin}(N_w - 2), \boldsymbol{X})) - \hat{w}(\mathsf{bin}(N_w - 1), \boldsymbol{X}) = 0$ for all $\boldsymbol{X} \in \{0,1\}^n$.

Indexer and Preprocessing. Both P and V can compute $\hat{p}$ independently in the preprocessing phase since they have access to $\boldsymbol{p}$. Moreover, given the circuit C, the PIOP Indexer can initialize $\hat{q}$ and $\hat{\pi}$ and send them to P, and similarly send $[\![\hat{q}]\!]$ and $[\![\hat{\pi}]\!]$ to V.

HyperPlonk PIOP. Chen et al. [CBBZ23] propose the following PIOP for Definition 5.20:

PIOP 7: HYPERPLONK

Indexer: Initialize $\hat{q}$ and $\hat{\pi}$ and send them to P. Also send $[\![\hat{q}]\!]$ and $[\![\hat{\pi}]\!]$ to V.

Protocol: $\langle \mathsf{P}(\mathbb{i}_{\mathsf{HPC}}, \mathbb{x}_{\mathsf{HPC}}, \mathbb{w}_{\mathsf{HPC}}), \mathsf{V}(\mathbb{i}_{\mathsf{HPC}}, \mathbb{x}_{\mathsf{HPC}}, [\![\hat{q}]\!], [\![\hat{\pi}]\!]) \rangle$:

1. P initializes $[\![\hat{w}]\!]$ and sends it to V.
2. Gate identity check: P and V invoke the zerocheck PIOP for the claim "$\hat{f}(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$".
3. Wiring identity check: P and V invoke the permcheck PIOP for the claim "$\hat{w}(\boldsymbol{x}) = \hat{w}(\pi(\boldsymbol{x}))$ for all $\boldsymbol{x} \in \{0,1\}^{n+n_w}$".
4. Public input consistency check: V samples $\boldsymbol{r} \xleftarrow{\$} \mathbb{F}^{n_p}$ and checks $\hat{p}(\boldsymbol{r}) = \hat{w}(0^{n+n_w-n_p}, \boldsymbol{r})$ by making an oracle query to $\hat{p}$ and $\hat{w}$.

We construct a read-write streaming prover for the above PIOP:

Read-write Streaming Algorithm 8: HYPERPLONK RELATION

$\mathsf{P}(\mathtt{i}_{\mathsf{HPC}}, \mathbb{x}_{\mathsf{HPC}}, \mathbf{R}_q(\hat{q}), \mathbf{R}_w(\hat{w}), \mathbf{R}_\pi(\hat{\pi}))$:

1. Create $[\![\hat{w}]\!]$ and send it to V.

Split selector polynomial into N_q streams: a stream for each type of gate:

2. For i in $[0, \ldots, N_q - 1]$: $\mathbf{S}_{q,i}.\mathsf{init}(N)$.
3. $(\mathbf{S}_{q,0}, \ldots, \mathbf{S}_{q,N_q-1}) \leftarrow \mathsf{SplitLSB}(\hat{q}, n_q)$.

Split witness polynomial into N_w streams so that the i-th stream stores the i-th input of all gates and the $(N_w - 1)$-th stream stores the output of all gates:

4. For i in $[0, \ldots, N_w - 1]$: $\mathbf{S}_{w,i}.\mathsf{init}(N)$.
5. $(\mathbf{S}_{w,0}, \ldots, \mathbf{S}_{w,N_w-1}) \leftarrow \mathsf{SplitLSB}(\hat{w}, n_w)$.

Using PToS, initialize sumcheck instance of the permcheck for the wiring identity:

6. $(\mathtt{i}_1 = (\_, \_, k_1, h_1), \mathbb{x}_1, \mathbb{w}_1) \leftarrow \mathsf{PToS}((\mathbb{F}, n + n_w, \pi), ([\![\hat{w}]\!], [\![\hat{w}]\!], [\![\hat{\pi}]\!]), (\mathbf{R}_w, \mathbf{R}_w))$.

Using ZToS initialize sumcheck instance of the zerocheck for the gate identity:

7. Using the combining gate function $\tilde{f}$, define $\mathtt{i}_{\mathsf{ZC}} := (\mathbb{F}, n, N_w + N_q, \tilde{f})$
8. Define $\mathbb{x}_{\mathsf{ZC}} := ([\![q_0]\!], \ldots, [\![q_{N_q-1}]\!], [\![w_0]\!], \ldots, [\![w_{N_w-1}]\!])$.
9. Define $\mathbb{w}_{\mathsf{ZC}} := (\mathbf{S}_{q,0}, \ldots, \mathbf{S}_{q,N_q-1}, \mathbf{S}_{w,0}, \ldots, \mathbf{S}_{w,N_w-1})$.
10. $(\mathtt{i}_2 = (\_, \_, k_2, h_2), \mathbb{x}_2, \mathbb{w}_2) \leftarrow \mathsf{ZToS}(\mathtt{i}_{\mathsf{ZC}}, \mathbb{x}_{\mathsf{ZC}}, \mathbb{w}_{\mathsf{ZC}})$.

Batch the two sumcheck instances together:

11. Define $\mathtt{i} := (\mathbb{F}, n, k_1 + k_2, 2, h_1, h_2)$ and $\mathbb{x} := (\mathbb{x}_1, \mathbb{x}_2)$ and $\mathbb{w} = (\mathbb{w}_1, \mathbb{w}_2)$
12. P and V invoke a RW streaming batch sumcheck PIOP for sumprod polynomials for $(\mathtt{i}, \mathbb{x}, \mathbb{w})$ as defined above.

Lemma 5.22. *PIOP 7 and Algorithm 8 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{HPC}}$ with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(k \cdot \ell \cdot N)$	$O(k + \ell + \log N)$	$O(kN)$	$O(k)$	$O(kN/\lvert\mathbb{F}\rvert)$	$O(k \log N)$	$O(k)$

where $k = d \cdot (N_q + N_w)$ and ℓ is the number of monomials required to represent the combining gate function $\tilde{f}$.

Proof. The efficiency parameters of Algorithm 8 follow from the efficiency parameters of the permcheck and zerocheck PIOPs (Lemma 5.17 and Lemma 5.9).

Algorithm 8 assumes without loss of generality that $\tilde{f}$ is a multilinear polynomial ($d = 1$), which enables the invocation of the RW streaming zerocheck PIOP on it. This assumption is equivalent to the assumption that all types of gates are multilinear polynomials. If a gate has degree $d > 1$ with arity $N_w - 1$, then it can be represented as a multilinear polynomial in $d \cdot (N_w - 1)$ variables whose inputs can be obtained by creating d copies of each input stream defined in Step 2 and Step 4.

Given the aforementioned generalization to higher degree gates, there are $d \cdot (N_q + N_w)$ number of input RW streams for the zerocheck PIOP call, each of length N. Therefore, the time, random-access space, and streaming space required to invoke the RW streaming zerocheck PIOP are $O(k \cdot \ell \cdot N), O(k + \ell + \log N), O(k \cdot N)$ respectively, where k, ℓ are defined in the lemma statement.

Moreover, $\hat{w}$ has size $d \cdot N_w \cdot N$. Therefore, the time, random-access space, and streaming space required to invoke the RW streaming permcheck PIOP are $O(d \cdot N_w \cdot N), O(d \cdot N_w + \log N), O(d \cdot N_w \cdot N)$ respectively.

The remaining PIOP properties follow from analysis of Chen et al. [CBBZ23]. □

6 Streaming polynomial commitment schemes

In this section we recall the definition of polynomial commitment (PC) schemes and present our read-write streaming algorithms for the PC scheme of Papamanthou, Shi, and Tamassia [PST13]. We defer to Appendices B and C our RW streaming algorithms for other PC schemes based on inner product arguments.

6.1 Multilinear polynomial commitment schemes

A multilinear polynomial commitment scheme is a tuple of algorithms $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Check})$ with the following syntax.

- $\mathsf{PC.Setup}(1^\lambda, n) \to (\mathsf{ck}, \mathsf{vk})$. On input a security parameter λ (in unary), and a maximum number of variables $n \in \mathbb{N}$, $\mathsf{PC.Setup}$ samples committer and verifier keys ck and vk.
- $\mathsf{PC.Commit}(\mathsf{ck}, p) \to C$. On input ck and a multilinear polynomial p over the field $\mathbb{F}$, $\mathsf{PC.Commit}$ outputs a commitment C to p.
- $\mathsf{PC.Open}(\mathsf{ck}, (C, \boldsymbol{z}, y), p) \to \pi$. On input ck, a polynomial p, a commitment to it C, an evaluation point $\boldsymbol{z}$, and a claimed evaluation y, $\mathsf{PC.Open}$ outputs an evaluation proof π asserting that y is indeed the evaluation of p at $\boldsymbol{z}$.
- $\mathsf{PC.Check}(\mathsf{vk}, (C, \boldsymbol{z}, y), \pi) \to b \in \{0, 1\}$. On input vk, a commitment C, an evaluation point $\boldsymbol{z}$, a claimed evaluation y, and an evaluation proof π, $\mathsf{PC.Check}$ outputs 1 if π attests that the polynomial p committed in C evaluates to y at $\boldsymbol{z}$.

A polynomial commitment scheme PC must satisfy standard completeness and extractability properties, but we do not recall these definitions here, as we only construct read-write streaming versions for the Commit and Open algorithms of existing schemes. We thus cannot affect extractability, and we show that completeness is preserved by demonstrating that our algorithms produce the same output as their non-streaming counterparts.

6.2 The PST13 PC scheme

We now recall the polynomial commitment scheme of Papamanthou, Shi, and Tamassia [PST13] as presented in [XZZPS19].

$\mathsf{PST.Setup}(1^\lambda, n) \to (\mathsf{ck}, \mathsf{vk})$:
1. Obtain $\langle \mathsf{group} \rangle = (\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G, H) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
2. Sample random $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \xleftarrow{\$} \mathbb{F}^n$.
3. For each j in $[0, 1, \ldots, n]$:
4. Set $\mathsf{ck}_j := [\mathrm{eq}_j(\boldsymbol{\alpha}_{[n-j+1:n]}, \boldsymbol{i}) \cdot G]_{\boldsymbol{i} \in \{0,1\}^j}$.
5. Set $\mathsf{ck} := ([\mathsf{ck}_j]_{j=0}^n, \langle \mathsf{group} \rangle)$.
6. Set $\mathsf{vk} := ([\alpha_i \cdot H]_{i \in [n]}, \langle \mathsf{group} \rangle)$.
7. Output $(\mathsf{ck}, \mathsf{vk})$.

$\mathsf{PST.Commit}(\mathsf{ck}, \boldsymbol{p}) \to C_p$:
1. Parse ck as $([\mathsf{ck}_j]_{j=0}^n, \langle \mathsf{group} \rangle)$.
2. Parse ck_n as $[\mathrm{eq}_n(\boldsymbol{\alpha}, \boldsymbol{i}) \cdot G]_{\boldsymbol{i} \in \{0,1\}^n}$.
3. Output $C_p := \sum_{\boldsymbol{i} \in \{0,1\}^n} p_i \cdot \mathrm{eq}_n(\boldsymbol{\alpha}, \boldsymbol{i}) \cdot G = \langle \boldsymbol{p}, \mathsf{ck}_n \rangle$.

$\mathsf{PST.Open}(\mathsf{ck}, (C_p, \boldsymbol{z}, y), \boldsymbol{p}) \to \pi$:
Parse: $\mathsf{ck} = ([\mathsf{ck}_j]_{j=0}^n, \langle \mathsf{group} \rangle)$.
1. For each j in $[1, \ldots, n]$:
2. Compute $\boldsymbol{q}_j$ corresponding to $q_j(\boldsymbol{X})$ such that $p(\boldsymbol{X}) - y = \sum_{i=1}^n q_i(\boldsymbol{X}) \cdot (X_i - z_i)$.
3. Compute $\pi_j := q_j(\boldsymbol{\alpha}) \cdot G = \langle \boldsymbol{q}_j, \mathsf{ck}_{n-j} \rangle$.
4. Output evaluation proof $\pi := (\pi_1, \ldots, \pi_n)$.

$\mathsf{PST.Check}(\mathsf{vk}, (C_p, \boldsymbol{z}, y), \pi) \to \{0, 1\}$:
Parse: $\mathsf{vk} = ([\alpha_i \cdot H]_{i \in [n]}, \langle \mathsf{group} \rangle)$ and $\pi = (\pi_1, \ldots, \pi_n)$.
1. Accept if $e(C_p - y \cdot G, H) = \sum_{i=1}^n e(\pi_i, (\alpha_i - z_i) \cdot H)$.

6.2.1 Read-write streaming algorithms for PST13

Our read-write streaming algorithms for PST.Commit and PST.Open are described below:

Read-write Streaming Algorithm 9: PST.Commit

$\mathsf{PST.Commit}(\mathsf{ck}, \mathbf{R}_p(p))$:
Parse: $\mathsf{ck} = ([\mathbf{R}_{\mathsf{ck}_j}(\mathsf{ck}_j)]_{j=0}^n, \langle \mathsf{group} \rangle)$.
1. Output $C_p \leftarrow \mathsf{InnerProd}(\mathbf{R}_p, \mathbf{R}_{\mathsf{ck}_n})$.

Read-write Streaming Algorithm 10: PST.Open

$\mathsf{PST.Open}(\mathsf{ck}, (C_p, \boldsymbol{z}, y), \mathbf{R}_p(p))$:
Parse: $\mathsf{ck} = ([\mathbf{R}_{\mathsf{ck}_j}(\mathsf{ck}_j)]_{j=0}^n, \langle \mathsf{group} \rangle)$.
1. For each i in $[1, \ldots, n]$:
2. Obtain $(\mathbf{R}_g, \mathbf{R}_h) \leftarrow \mathsf{SplitLR}(\mathbf{R}_p)$.
3. $\mathbf{R}_h \leftarrow \mathsf{LinComb}(1, -1, \mathbf{R}_h, \mathbf{R}_g)$.
4. Compute the commitment $\pi_i \leftarrow \mathsf{InnerProd}(\mathbf{R}_h, \mathbf{R}_{\mathsf{ck}_{n-i}})$.
5. $\mathbf{R}_g \leftarrow \mathsf{LinComb}(1, z_i, \mathbf{R}_g, \mathbf{R}_h)$.
6. Set $\mathbf{R}_p \leftarrow \mathbf{R}_g$.
7. Output $\pi := (\pi_1, \ldots, \pi_n)$.

Lemma 6.1. *Algorithm 9 and Algorithm 10 are read-write streaming algorithms for* PST.Commit *and* PST.Open *respectively with the following efficiency when committing to n-variate multilinear polynomials (where $N = 2^n$):*
- *Algorithm 9 requires $O(N)$ running time, $O(\log N)$ random-access space and $O(1)$ streaming space.*
- *Algorithm 10 requires $O(N)$ running time, $O(\log N)$ random-access space and $O(N)$ streaming space.*

Proof. It is easy to inspect that Algorithm 9 realizes PST.Commit and requires $O(N)$ group operations and $O(\log N)$ random-access space.

We now verify that Algorithm 10 implements the same function as PST.Open by writing out what Algorithm 10 is computing in its streams. Setting $r_0(X_1, \ldots, X_n) := p(X_1, \ldots, X_n)$, PST.Open recursively defines $r_j(X_{j+1}, \ldots, X_n)$ and $q_j(X_{j+1}, \ldots, X_n)$ for each $j \in [n]$ as follows:

$$\begin{aligned} r_{j-1}(X_j, \ldots, X_n) &= g_j(X_{j+1}, \ldots, X_n) + X_j \cdot h_j(X_{j+1}, \ldots, X_n) \\ &= (g_j(X_{j+1}, \ldots, X_n) + z_j \cdot h_j(X_{j+1}, \ldots, X_n)) + (X_j - z_j) \cdot h_j(X_{j+1}, \ldots, X_n) \\ &:= r_j(X_{j+1}, \ldots, X_n) + (X_j - z_j) \cdot q_j(X_{j+1}, \ldots, X_n) \end{aligned}$$

It then commits to each $q_i(\boldsymbol{X})$ as $\pi_i := q_i(\boldsymbol{\alpha}) \cdot G = \langle \boldsymbol{q}_j, \mathsf{ck}_{n-j} \rangle$.

Algorithm 10 identically obtains evaluations of these polynomials over their respective boolean hypercubes as follows: for every point $\boldsymbol{i} \in \{0, 1\}^{n-j+1}$, if $i_1 = 0$, we have $g_j(i_2, \ldots, i_{n-j+1}) = r_{j-1}(0, i_2, \ldots, i_{n-j+1})$.

If $i_1 = 1$, then $r_{j-1}(1, i_2, \ldots, i_{n-j+1}) = g_j(i_2, \ldots, i_{n-j+1}) + h_j(i_2, \ldots, i_{n-j+1})$. Thus, the algorithm sets $h_j(i_2, \ldots, i_{n-j+1}) = r_{j-1}(1, i_2, \ldots, i_{n-j+1}) - r_{j-1}(0, i_2, \ldots, i_{n-j+1})$. It then defines $r_j(i_2, \ldots, i_{n-j+1}) := g_j(i_2, \ldots, i_{n-j+1}) + z_j \cdot h_j(i_2, \ldots, i_{n-j+1})$ and $q_j(i_2, \ldots, i_{n-j+1}) := h_j(i_2, \ldots, i_{n-j+1})$.

Algorithm 10 will now need to commit to the witness polynomials $q_1(\boldsymbol{X}), \ldots, q_n(\boldsymbol{X})$. The commitment to the polynomial $q_j(X_{j+1}, \ldots, X_n)$ can be computed as

$$q_j(\alpha_{j+1}, \ldots, \alpha_n) \cdot G = \sum_{\boldsymbol{i} \in \{0,1\}^{n-j}} q(\boldsymbol{i}) \cdot \mathrm{eq}_{n-j}(\boldsymbol{\alpha}_{[j+1,n]}, \boldsymbol{i}) \cdot G$$

This is realised by computing $\pi_j := \mathsf{InnerProd}(\mathbf{R}_{q_j}, \mathbf{R}_{\mathsf{ck}_{n-j}})$.

Efficiency analysis. We now inspect the efficiency of Algorithm 10. In the j-th round of the for loop, the algorithm computes $h_j(X_{j+1}, \ldots, X_n)$ and $h_j(X_{j+1}, \ldots, X_n)$ given $r_{j-1}(X_j, \ldots, X_n)$. It then sets $q_j(X_{j+1}, \ldots, X_n) := h_j(X_{j+1}, \ldots, X_n)$ and $r_j(X_{j+1}, \ldots, X_n) := g_j(X_{j+1}, \ldots, X_n) + z_j \cdot h_j(X_{j+1}, \ldots, X_n)$.

To do this, the RW streaming algorithm starts by setting $\boldsymbol{R} := \boldsymbol{p}$, which is a vector of length 2^n. In the j-th step of the recursion, the length of $\boldsymbol{R}$ is 2^{n-j+1}. It sets $\boldsymbol{g} := \boldsymbol{R}_L$ and $\boldsymbol{h} := \boldsymbol{R}_R - \boldsymbol{R}_L$, which takes $O(2^{n-j+1})$ time. It then proceeds to set $\boldsymbol{R} \leftarrow \boldsymbol{g} + z_j \cdot \boldsymbol{h}$, which takes $O(2^{n-j})$ time, and commits to $\boldsymbol{h}$, which requires $O(2^{n-j})$ group operations. In total, the j-th round takes $O(2^{n-j+1})$ time.

Thus the entire protocol thus takes $O(2^n) = O(N)$ time. Since all the inputs are provided as streams, the algorithm only requires $O(n) = O(\log N)$ space to keep track of its position in the stream.

□

A An alternative PIOP for permcheck

Recall that the HyperPlonk PIOP invokes a permcheck PIOP (Section 5.3.4) to check the wiring constraints of the circuit. In this section we present a concrete optimization for the permcheck PIOP, which we call the *split permcheck* PIOP. In the split permcheck PIOP, the witness polynomials p and q are not available as single multilinear polynomials, but are *split* into ν multilinear polynomials each. Additionally, there are ν permutations $\pi^{(1)}, \ldots, \pi^{(\nu)}$, and the prover wants to prove to the verifier that for all $i \in [\nu]$ and $\boldsymbol{x} \in \{0,1\}^n$, $p^{(i)}(\boldsymbol{x}) = q^{(i)}(\pi^{(i)}(\boldsymbol{x}))$. We discuss how this PIOP is useful in proving the wiring constraint in Appendix A.4.

To build the split permcheck PIOP, we first build a PIOP for sumcheck for *rational functions* in Appendix A.1, use it to obtain the *split multiset-equality-check* PIOP in Appendix A.2 using techniques similar to [Hab22], and then use the split multiset-equality-check PIOP to build the split permcheck PIOP in Appendix A.3.

A.1 Sumcheck for rational functions

We begin by defining rational functions:

Definition A.1 (Rational function)**.** *An n-variate (d, ℓ)-rational function is a function f such that*

$$f(\boldsymbol{X}) = \frac{p(\boldsymbol{X})}{q(\boldsymbol{X})} = \frac{h_p\left(p^{(1)}(\boldsymbol{X}), \ldots, p^{(d)}(\boldsymbol{X})\right)}{h_q\left(q^{(1)}(\boldsymbol{X}), \ldots, q^{(d)}(\boldsymbol{X})\right)} \ ,$$

where p, q are (d, ℓ)-sumprod polynomials, $p^{(i)}, q^{(i)}$ are multilinear polynomials for all $i \in [d]$, and h_p, h_q are multilinear polynomials.

We can now define the relation for sumcheck for rational functions:

Definition A.2 (Sumcheck for rational functions)**.** *This relation $\mathscr{R}_{\mathsf{MRSC}}$ contains tuples of the form*

$$\begin{aligned}(\mathbb{i}_{\mathsf{MRSC}}, \mathbb{x}_{\mathsf{MRSC}}, \mathbb{w}_{\mathsf{MRSC}}) =& ((\mathbb{F}, n, d, h_p, h_q), (\llbracket p_1 \rrbracket, \ldots, \llbracket p_d \rrbracket, \llbracket q_1 \rrbracket, \ldots, \llbracket q_d \rrbracket, \sigma), \\ & (p_1, \ldots, p_d, q_1, \ldots, q_d))\end{aligned}$$

such that $\sum_{\boldsymbol{x} \in \{0,1\}^n} \hat{h}_p(\boldsymbol{x}) / \hat{h}_q(\boldsymbol{x}) = \sigma$ where $\hat{h}_p, \hat{h}_q$ are (d, ℓ)-sumprod polynomials. In particular, $\hat{h}_p(\boldsymbol{X}) = h_p\left(p^{(1)}(\boldsymbol{X}), \ldots, p^{(d)}(\boldsymbol{X})\right)$ and $\hat{h}_q(\boldsymbol{X}) = h_q\left(q^{(1)}(\boldsymbol{X}), \ldots, q^{(d)}(\boldsymbol{X})\right)$.

The following is a PIOP for $\mathscr{R}_{\mathsf{MRSC}}$:

PIOP 8: SUMCHECK for (d, ℓ)-rational functions

$\langle \mathsf{P}(\mathbb{i}_{\mathsf{MRSC}}, \mathbb{x}_{\mathsf{MRSC}}, \mathbb{w}_{\mathsf{MRSC}}), \mathsf{V}(\mathbb{i}_{\mathsf{MRSC}}, \mathbb{x}_{\mathsf{MRSC}}) \rangle$:

1. P computes $f : \{0,1\}^n \to \mathbb{F}$ such that $f(\boldsymbol{X}) = \hat{h}_p(\boldsymbol{X}) \cdot \hat{h}_q(\boldsymbol{X})^{-1}$.
2. P sends $\llbracket \hat{f} \rrbracket$ to V where $\hat{f}$ is the multilinear extension of f.
3. P and V invoke a zerocheck PIOP for the polynomial $(\hat{h}_q \cdot \hat{f} - \hat{h}_p) = 0$.
4. P and V invoke a sumcheck PIOP for multilinear polynomials for the claim "$\sum_{\boldsymbol{x} \in \{0,1\}^n} \hat{f}(\boldsymbol{x}) = \sigma$".

The key idea of the above protocol is as follows:

1. Obtain the multilinear polynomial $\hat{f}(\boldsymbol{X})$ which is equal to $\hat{h}_p(\boldsymbol{X}) \cdot \hat{h}_q(\boldsymbol{X})^{-1}$ on the boolean hypercube.
2. P commits to $\hat{f}$ and sends $\llbracket \hat{f} \rrbracket$ to V.
3. In order to prove to V that $\hat{f}(\boldsymbol{x}) = p(\boldsymbol{x})/q(\boldsymbol{x})$ for all $\boldsymbol{x} \in \{0,1\}^n$, P proves that $\hat{f}(\boldsymbol{x}) \cdot q(\boldsymbol{X}) - p(\boldsymbol{X}) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$, which is achieved by engaging in a zerocheck PIOP.

4. Finally, since $\hat{f}$ is multilinear, P and V can engage in a sumcheck protocol on $\hat{f}$.

We now present a RW streaming prover for $\mathscr{R}_{\mathsf{MRSC}}$:

Read-write Streaming Algorithm 11: SUMCHECK for (d,ℓ)-rational functions

$\mathsf{P}(\mathtt{i}_{\mathsf{MRSC}}, \mathbb{x}_{\mathsf{MRSC}}, (\mathbf{R}_p^{(1)}(p^{(1)}), \ldots, \mathbf{R}_p^{(d)}(p^{(d)}), \mathbf{R}_q^{(1)}(q^{(1)}), \ldots, \mathbf{R}_q^{(d)}(q^{(d)}))):$

Obtain read-only stream of evaluations of $\hat{f}$ over $\{0,1\}^n$:

1. $\mathbf{S}_p \rightsquigarrow \mathsf{Map}(h_p) \rightsquigarrow \mathsf{Zip} \rightsquigarrow (\mathbf{R}_p^{(1)}, \ldots, \mathbf{R}_p^{(d)})$.
2. $\mathbf{S}_q \rightsquigarrow \mathsf{Map}(h_q) \rightsquigarrow \mathsf{Zip} \rightsquigarrow (\mathbf{R}_q^{(1)}, \ldots, \mathbf{R}_q^{(d)})$.
3. $\mathbf{S}_f \rightsquigarrow \mathsf{Map}(/) \rightsquigarrow \mathsf{Zip} \rightsquigarrow (\mathbf{S}_p, \mathbf{S}_q)$.
4. Create the oracle $[\![\hat{f}]\!]$ using $\mathbf{S}_f$ and send it to V.

Obtain sumcheck claim for the zerocheck using ZToS:

5. Define $t_0(X_1, \ldots, X_k, Y_1, \ldots, Y_k, Z) := h_q(Y_1, \ldots, Y_k) \cdot Z - h_p(X_1, \ldots, X_k)$.
6. $\mathtt{i}_z := (\mathbb{F}, n, d+1, t_0)$
7. $\mathbb{x}_z := ([\![p^{(1)}]\!], \ldots, [\![p^{(d)}]\!], [\![q^{(1)}]\!], \ldots, [\![q^{(d)}]\!], [\![\hat{f}]\!])$
8. $\mathbb{w}_z := (\mathbf{R}_p^{(1)}, \ldots, \mathbf{R}_p^{(d)}, \mathbf{R}_q^{(1)}, \ldots, \mathbf{R}_q^{(d)}, \mathbf{S}_f)$
9. $(\mathtt{i}_z = (\_, \_, \_, t_1), \mathbb{x}_z, \mathbb{w}_z) \leftarrow \mathsf{ZToS}(\mathtt{i}_z, \mathbb{x}_z, \mathbb{w}_z)$.

Batch the sumcheck instance above with the sumcheck over $\hat{f}$:

10. Define identity function $\mathsf{id}(X) := X$.
11. Define $\mathtt{i} := (\mathbb{F}, n, d+1, 2, \mathsf{id}, t_1)$ and $\mathbb{x} := ((\![\hat{f}]\!], \sigma), \mathbb{x}_z)$ and $\mathbb{w} := ((\mathbf{S}_f), \mathbb{w}_z)$
12. P and V invoke a RW streaming PIOP for a batch of sumprod polynomials for $(\mathtt{i}, \mathbb{x}, \mathbb{w})$ as defined above.

Lemma A.3. *PIOP 8 and Algorithm 11 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{MRSC}}$ for n-variate (d,ℓ)-rational functions with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(d\ell N)$	$O(d+\ell+\log N)$	$O(dN)$	$O(d)$	$O(dN/\lvert\mathbb{F}\rvert)$	$O(d \log N)$	$O(d)$

Proof. Time and space efficiency properties follow from the analysis of RW streaming zerocheck for (d,ℓ)-sumprod polynomials. The prover is complete by construction, and the soundness error is $O(dN/|\mathbb{F}|)$ because the degree of f is $O(dN)$. □

A.2 Split multiset-equality-check

The following PIOP is inspired from the multiset-equality-check of [Hab22]. Given a split size ν, multilinear polynomials $p^{(1)}, \ldots, p^{(\nu)}$ and multilinear polynomials $q^{(1)}, \ldots, q^{(\nu)}$, P wants to prove to V that

$$\bigcup_{j=1}^{\nu} \bigcup_{\boldsymbol{x} \in \{0,1\}^n} \left\{\!\!\left\{ p^{(j)}(\boldsymbol{x}) \right\}\!\!\right\} = \bigcup_{j=1}^{\nu} \bigcup_{\boldsymbol{x} \in \{0,1\}^n} \left\{\!\!\left\{ q^{(j)}(\boldsymbol{x}) \right\}\!\!\right\} \quad . \tag{1}$$

This is identical to the multiset-equality-check in Section 5.3.3, except that polynomials p and q have been split into ν parts. In the RW streaming setting this represents the fact they arrive in separate RW streams. Also, note that this is not the same as a *batch* multiset-equality-check where one would have ν separate equalities, and can be reduced to a sumcheck PIOP for a batch of sumprod polynomials Section 5.2.3.

This is formalized by the relation $\mathscr{R}_{\mathsf{SMC}}$, which contains tuples of the form

$$(\mathtt{i}_{\mathsf{SMC}}, \mathbb{x}_{\mathsf{SMC}}, \mathbb{w}_{\mathsf{SMC}}) = ((\mathbb{F}, n, \nu), ([\![p^{(1)}]\!], \ldots, [\![p^{(\nu)}]\!], [\![q^{(1)}]\!], \ldots, [\![q^{(\nu)}]\!]), (p^{(1)}, \ldots, p^{(\nu)}, q^{(1)}, \ldots, q^{(\nu)}))$$

such that Eq. (1) is satisfied. The following is a PIOP for $\mathscr{R}_{\mathsf{SMC}}$:

PIOP 9: SPLIT MULTISET-EQUALITY-CHECK

$\langle \mathsf{P}(\mathbb{i}_{\mathsf{SMC}}, \mathbb{x}_{\mathsf{SMC}}, \mathbb{w}_{\mathsf{SMC}}), \mathsf{V}(\mathbb{i}_{\mathsf{SMC}}, \mathbb{x}_{\mathsf{SMC}}) \rangle$:

1. V sends a random challenge $\alpha \in \mathbb{F}$ to P.
2. P and V engage in a sumcheck PIOP for rational functions for the claim "$\sum_{\boldsymbol{x} \in \{0,1\}^n} \sum_{i=1}^{\nu} (\alpha + p^{(i)}(\boldsymbol{x}))^{-1} - (\alpha + q^{(i)}(\boldsymbol{x}))^{-1} = 0$".

We describe a RW streaming prover that uses $O(\nu N)$ time and $O(\nu + \log N)$ random-access space to execute this PIOP:

Read-write Streaming Algorithm 12: SPLIT MULTISET-EQUALITY-CHECK

$\mathsf{P}(\mathbb{i}_{\mathsf{SMC}}, \mathbb{x}_{\mathsf{SMC}}, (\mathbf{R}_p^{(1)}(p^{(1)}), \ldots, \mathbf{R}_p^{(\nu)}(p^{(\nu)}), \mathbf{R}_q^{(1)}(q^{(1)}), \ldots, \mathbf{R}_q^{(\nu)}(q^{(\nu)})))$:

1. Receive a random challenge $\alpha \in \mathbb{F}$ from V.
2. Define function add: $\mathsf{add}(x) := x + \alpha$.
3. For i in $[1, \ldots, \nu]$:
4. $\quad \mathbf{S}_p^{(i)} \rightsquigarrow \mathsf{Map}(\mathsf{add}) \rightsquigarrow \mathbf{R}_p^{(i)}$.
5. $\quad \mathbf{S}_q^{(i)} \rightsquigarrow \mathsf{Map}(\mathsf{add}) \rightsquigarrow \mathbf{R}_q^{(i)}$.
6. Define $f_1(x_1, \ldots, x_\nu, y_1, \ldots, y_\nu) := \sum_{i=1}^{\nu} (-1)^{\nu-1} \prod_{j=1}^{\nu} x_j \cdot \prod_{j \neq i} y_j + \sum_{i=1}^{\nu} (-1)^{\nu} \prod_{j=1}^{\nu} y_j \cdot \prod_{j \neq i} x_j$.
7. Define $f_2(x_1, \ldots, x_\nu, y_1, \ldots, y_\nu) := \prod_{j=1}^{\nu} x_j \cdot y_j$.
8. Define $\mathbb{i} := (\mathbb{F}, n, 2\nu, f_1, f_2)$ and $\mathbb{x} := (\mathbb{x}_{\mathsf{SMC}}, 0)$ and $\mathbb{w} := (\mathbf{S}_p^{(1)}, \ldots, \mathbf{S}_p^{(\nu)}, \mathbf{S}_q^{(1)}, \ldots, \mathbf{S}_q^{(\nu)})$.
9. P and V invoke a RW streaming sumcheck PIOP for rational functions on $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ as defined above.

The foregoing RW streaming prover is executing a sumcheck PIOP for the following rational function:

$$\sum_{j=1}^{\nu} \sum_{i=0}^{N-1} \frac{1}{p_i^{(j)} + \alpha} = \sum_{j=1}^{\nu} \sum_{i=0}^{N-1} \frac{1}{q_i^{(j)} + \alpha}$$

$$\iff \sum_{i=0}^{N-1} \frac{\sum_{k=1}^{\nu} \prod_{j \neq k} (p_i^{(j)} + \alpha)}{\prod_{j=1}^{\nu} (p_i^{(j)} + \alpha)} = \sum_{i=0}^{N-1} \frac{\sum_{k=1}^{\nu} \prod_{j \neq k} (q_i^{(j)} + \alpha)}{\prod_{j=1}^{\nu} (q_i^{(j)} + \alpha)}$$

At first glance, it looks like the numerator has ν terms, where each term has degree $\nu - 1$, and would therefore require $O(\nu^2)$ time to compute. However, in the sumcheck for rational functions, these terms can be computed in $O(\nu)$ time by dividing the product $\prod_{j=1}^{\nu} (p_i^{(j)} + \alpha)$ by $(p_i^{(k)} + \alpha)$ to get the kth term of the numerator.

Lemma A.4. *PIOP 9 and Algorithm 12 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{SMC}}$ with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(\nu N)$	$O(\nu + \log N)$	$O(\nu N)$	$O(\nu)$	$O(\nu N / \lvert\mathbb{F}\rvert)$	$O(\nu \log N)$	$O(\nu)$

Proof. All properties follow from the analysis of RW streaming sumcheck for (d, ℓ)-rational functions. □

A.3 Split permcheck

Given a split size ν, multilinear polynomials $p^{(1)}, \ldots, p^{(\nu)}, q^{(1)}, \ldots, q^{(\nu)}$, and permutations $\pi_1, \ldots, \pi^{(\nu)}$, a prover P of the split permcheck PIOP wants to prove to V that

$$p^{(i)}(\boldsymbol{x}) = q^{(i)}(\pi^{(i)}(\boldsymbol{x})) \tag{2}$$

for all $i \in [\nu]$ and $\boldsymbol{x} \in \{0,1\}^n$. This is formalized by the relation $\mathscr{R}_{\mathsf{SPC}}$, which contains tuples of the form

$$(\mathtt{i}_{\mathsf{SPC}}, \mathtt{x}_{\mathsf{SPC}}, \mathtt{w}_{\mathsf{SPC}}) = ((\mathbb{F}, n, \nu, \pi^{(1)}, \ldots, \pi^{(\nu)}), (\llbracket p^{(1)} \rrbracket, \ldots, \llbracket p^{(\nu)} \rrbracket, \llbracket q^{(1)} \rrbracket, \ldots, \llbracket q^{(\nu)} \rrbracket), \\ (p^{(1)}, \ldots, p^{(\nu)}, q^{(1)}, \ldots, q^{(\nu)}, \hat{\pi}^{(1)}, \ldots, \hat{\pi}^{(\nu)}))$$

such that Eq. (2) is satisfied where $\hat{\pi}^{(i)}(\boldsymbol{x}) = \mathsf{int}(\pi^{(i)}(\boldsymbol{x}))$ for all $i \in [\nu]$ and $\boldsymbol{x} \in \{0,1\}^n$. The PIOP for $\mathscr{R}_{\mathsf{SPC}}$ is nearly identical to the one described for the permcheck relation in Section 5.3.4, except that the polynomials p and q are split into ν parts. We therefore omit this PIOP for brevity, and present the RW streaming prover for it:

Read-write Streaming Algorithm 13: SPLIT PERMCHECK

$\mathsf{P}(\mathtt{i}_{\mathsf{SPC}}, \mathtt{x}_{\mathsf{SPC}}, (\mathbf{R}_p^{(1)}(p^{(1)}), \ldots, \mathbf{R}_p^{(\nu)}(p^{(\nu)}), \mathbf{R}_q^{(1)}(q^{(1)}), \ldots, \mathbf{R}_q^{(\nu)}(q^{(\nu)}), \mathbf{R}_\pi^{(1)}(\hat{\pi}^{(1)}), \ldots, \mathbf{R}_\pi^{(\nu)}(\hat{\pi}^{(\nu)}))):$

1. Receive a random challenge $\beta \in \mathbb{F}$ from V.
2. Define function add: $\mathsf{add}(x, y) := x + \beta \cdot y$.
3. For j in $[1, \ldots, \nu]$:
4. $\quad \mathbf{S}_p^{(j)} \leftsquigarrow \mathsf{Map}(\mathsf{add}) \leftsquigarrow \mathsf{Zip} \leftsquigarrow (\mathbf{R}_p^{(j)}, \mathbf{R}_\pi^{(j)})$.
5. $\quad \mathbf{S}_q^{(j)} \leftsquigarrow \mathsf{Map}(\mathsf{add}) \leftsquigarrow \mathsf{Zip} \leftsquigarrow (\mathbf{R}_q^{(j)}, [i]_{i=1}^n)$.
6. Define $\mathtt{i} := (\mathbb{F}, n, \nu)$
7. Define $\mathtt{x} := (\llbracket p^{(1)} \rrbracket, \ldots, \llbracket p^{(\nu)} \rrbracket, \llbracket q^{(1)} \rrbracket, \ldots, \llbracket q^{(\nu)} \rrbracket)$.
8. Define $\mathtt{w} := (\mathbf{R}_p^{(1)}(p^{(1)}), \ldots, \mathbf{R}_p^{(\nu)}(p^{(\nu)}), \mathbf{R}_q^{(1)}(q^{(1)}), \ldots, \mathbf{R}_q^{(\nu)}(q^{(\nu)}))$.
9. P and V invoke a RW streaming split multiset-equality-check PIOP on $(\mathtt{i}, \mathtt{x}, \mathtt{w})$ as defined above.

Lemma A.5. *PIOP 6 and Algorithm 13 together comprise a read-write streaming PIOP for $\mathscr{R}_{\mathsf{SPC}}$ with the following properties:*

prover time	prover space			PIOP properties		
	random-access	streaming	# streams	soundness error	communication	# queries
$O(\nu N)$	$O(\nu + \log N)$	$O(\nu N)$	$O(\nu)$	$O(\nu N/\lvert\mathbb{F}\rvert)$	$O(\nu \log N)$	$O(\nu)$

Proof. All properties follow from the analysis of RW streaming PIOP for split multiset-equality-check. □

Remark A.6 (oracles of smaller polynomials for concrete efficiency)**.** The key concrete advantage of the split multiset-equality-check and split permcheck over the vanilla ones (Algorithm 6 and Algorithm 7) is that the multilinear polynomial $\hat{f}$ sent by P at Step 4 in Algorithm 11, has $O(N)$ terms instead of $O(\nu N)$ terms, which is a dominating factor in the concrete efficiency of the protocol. This is at the cost of slightly higher communication ($O(\nu \log N)$ instead of $O(\log(\nu N))$) and more polynomial queries ($O(\nu)$ instead of $O(1)$).

A.4 Using split permcheck for the wiring constraint

Recall the HyperPlonk wiring constraint for binary circuits from Section 2.2. The circuit C is represented by three vectors $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o} \in \mathbb{F}^N$ where $\boldsymbol{\ell}, \boldsymbol{r}$ are the left and right inputs to the gates, and $\boldsymbol{o}$ are the outputs of the gates. Furthermore, the wiring constraints of the circuit are represented by 2 permutation vectors $\boldsymbol{\pi}, \boldsymbol{\sigma} \in \mathbb{F}^N$ as follows: if the left input of gate i is the output of gate j, then $\pi_i = j$, and similarly, if the right input of gate i is the output of gate j then $\sigma_i = j$.

Let the multilinear extensions of these 5 vectors be $\hat{\ell}, \hat{r}, \hat{o}, \hat{\pi}, \hat{\sigma}$ respectively, and assume that P receives the stream of evaluations of these polynomials over the boolean hypercube in 5 separate RW streams. In order to prove that the wiring constraint is satisfied, P needs to do to prove that $\hat{\ell}(\boldsymbol{x}) = \hat{o}(\hat{\pi}(\boldsymbol{x}))$ and $\hat{r}(\boldsymbol{x}) = \hat{o}(\hat{\sigma}(\boldsymbol{x}))$

for all $\boldsymbol{x} \in \{0,1\}^n$. Therefore the HyperPlonk PIOP calls the split permcheck PIOP with the following arguments:

$$((\mathbb{F}, n, 2, \hat{\pi}, \hat{\sigma}), ([\![\hat{\ell}]\!], [\![\hat{r}]\!], [\![\hat{o}]\!], [\![\hat{o}]\!]), (\mathbf{R}_1(\hat{\ell}), \mathbf{R}_2(\hat{r}), \mathbf{R}_3(\hat{o}), \mathbf{R}_4(\hat{o}), \mathbf{R}_5(\hat{\pi}), \mathbf{R}_6(\hat{\sigma}))) \ ,$$

which improves the concrete efficiency of the wiring constraint check by approximately a factor of 2.

B Generalized inner product arguments

A core building block for many polynomial commitment schemes [BCCGP16; WTSTW18; Lee21; BMMTV21; BGH19; BCMS20] is an inner product argument (IPA) [BCCGP16; BBBPWM18] or its generalization [LMR19; BMMTV21].

As a stepping stone to constructing read-write streaming variants of these polynomial commitments, in this section we construct a read-write streaming prover that requires only logarithmic random access space for the generalized inner product argument (GIPA) of [BMMTV21]. At a high level, the latter is an argument that allows a prover to convince a verifier that $\langle \boldsymbol{x}, \boldsymbol{w} \rangle = v$, where $\boldsymbol{x}, \boldsymbol{w}$ are committed vectors, and v is the claimed result of a suitable inner-product between $\boldsymbol{x}$ and $\boldsymbol{w}$.

B.1 Commitment schemes

We recall the definition of doubly-homomorphic and inner-product commitment schemes [BMMTV21], along with several constructions of these that will appear in our instantiations of GIPA.

Definition B.1 (commitment scheme)**.** *A commitment scheme* $\mathsf{CM} = (\mathsf{Setup}, \mathsf{Commit})$ *over a universe of key spaces* $\{\mathcal{K}_i\}_{i\in\mathbb{N}}$ *message spaces* $\{\mathcal{M}_i\}_{i\in\mathbb{N}}$ *enables a party to generate a (perfectly) hiding and (computationally) binding commitment to a given message* $m \in \mathcal{M}$*.*

- Setup: *on input a security parameter and a description of the message space* $\mathcal{M}_i$*,* $\mathsf{CM.Setup}$ *samples a commitment key* $\mathsf{ck} \in \mathcal{K}_i$*.*
- Commit: *on input public parameters* ck*, message* $m \in \mathcal{M}_i$*, and randomness* r*,* $\mathsf{CM.Commit}$ *outputs a commitment* C_m *to* m*.*

Definition B.2 (Doubly homomorphic commitment)**.** *Let* $(\mathcal{K}, +)$*,* $(\mathcal{M}, +)$ *and* $(\mathsf{Image}(\mathsf{CM.Commit}), +)$ *define abelian groups. A commitment scheme* $\mathsf{CM} = (\mathsf{Setup}, \mathsf{Commit})$ *is doubly homomorphic if for all* $\mathsf{ck}_1, \mathsf{ck}_2 \in \mathcal{K}$ *and* $m_1, m_2 \in \mathcal{M}$ *we have*

1. $\mathsf{Commit}(\mathsf{ck}_1, m_1) + \mathsf{Commit}(\mathsf{ck}_2, m_1) = \mathsf{Commit}(\mathsf{ck}_1 + \mathsf{ck}_2, m_1)$
2. $\mathsf{Commit}(\mathsf{ck}_1, m_1) + \mathsf{Commit}(\mathsf{ck}_1, m_2) = \mathsf{Commit}(\mathsf{ck}_1, m_1 + m_2)$

In particular the above also implies $\mathsf{Commit}(\alpha \cdot \mathsf{ck}_1, m_1) = \alpha \cdot \mathsf{Commit}(\mathsf{ck}_1, m_1)$ *and* $\mathsf{Commit}(\mathsf{ck}_1, \alpha \cdot m_1) = \alpha \cdot \mathsf{Commit}(\mathsf{ck}_1, m_1)$*.*

Definition B.3 (Inner product map)**.** *A map* $\circledast : \mathcal{M}_1 \times \mathcal{M}_2 \to \mathcal{M}_3$ *from two groups of prime order* p *to a third group of order* p *is an inner product map if for all* $a, b \in \mathcal{M}_1$ *and* $c, d \in \mathcal{M}_2$ *we have that*

$$(a + b) \circledast (c + d) = a \circledast c + a \circledast d + b \circledast c + b \circledast d$$

Given an inner product $\circledast$ *between groups we define the inner product between vector spaces* $\langle \cdot, \cdot \rangle_{\circledast} : \mathcal{M}_1^N \times \mathcal{M}_2^N \to \mathcal{M}_3$ *to be* $\langle \boldsymbol{a}, \boldsymbol{b} \rangle_{\circledast} := \sum_{i=1}^{N} a_i \circledast b_i$*.*

Definition B.4 (Inner product commitment)**.** *Let* CM *be a doubly homomorphic commitment scheme with message space* $\mathcal{M} = \mathcal{M}_1^m \times \mathcal{M}_2^m \times \mathcal{M}_3$ *and key space* $\mathcal{K} = \mathcal{K}_1^m \times \mathcal{K}_2^m \times \mathcal{K}_3$ *defined for all* $m \in [2^j]_{j\in\mathbb{N}}$*, where* $|\mathcal{M}_i| = |\mathcal{K}_i| = p$ *is prime for* $i \in [3]$*. Let* $\circledast : \mathcal{M}_1 \times \mathcal{M}_2 \to \mathcal{M}_3$ *be an inner product map. We call*

$(\mathsf{CM}, \circledast)$ *an inner product commitment if there exist efficient deterministic functions* Split *and* Collapse *such that for all* $m \in [2^j]_{j\in\mathbb{N}}$, $M \in \mathcal{M}$, *and* $\mathsf{ck} \in \mathcal{K}$, *if* $\mathsf{Split}(\mathsf{ck}_i) = (\mathsf{ck}_i^L, \mathsf{ck}_i^R)$ *for* $i \in [2]$ *it holds that:*

$$\mathsf{Collapse}\left(\mathsf{CM}\left(\begin{array}{c|c} \mathsf{ck}_1 & M_1 \;||\; M_1 \\ \mathsf{ck}_2 & M_2 \;||\; M_2 \\ \mathsf{ck}_3 & M_3 \end{array}\right)\right) = \mathsf{CM}\left(\begin{array}{c|c} \mathsf{ck}_1^L + \mathsf{ck}_1^R & M_1 \\ \mathsf{ck}_2^L + \mathsf{ck}_2^R & M_2 \\ \mathsf{ck}_3 & M_3 \end{array}\right)$$

The above requirement is referred to as the collapsing property.

Constructions of inner-product commitment schemes. We recall the Pedersen commitment scheme CM_P which has message space $\{\mathbb{F}^N\}_{N\in\mathbb{N}}$, and the AFGHO commitment schemes [AFGHO16] CM_1 and CM_2, which have message spaces $\{\mathbb{G}_1^N\}_{N\in\mathbb{N}}$ and $\{\mathbb{G}_2^N\}_{N\in\mathbb{N}}$ respectively. For notational convenience, in this section we write $\bar{1}$ to denote the subscript 2, and write $\bar{2}$ to denote the subscript 1.

$\mathsf{CM}_P.\mathsf{Setup}(1^\lambda, N) \to \mathsf{ck}_P$: 1. Output $\mathsf{ck}_P := \mathbf{\Gamma}_1 \xleftarrow{\$} \mathbb{G}_1^N$.	$\mathsf{CM}_P.\mathsf{Commit}(\mathsf{ck}_P, \boldsymbol{x}) \to C_{\boldsymbol{x}}$: 1. Parse ck_P as $\mathbf{\Gamma}_1$. 2. Output $\langle \boldsymbol{x}, \mathbf{\Gamma}_1 \rangle_{\mathbb{G}}$.
$\mathsf{CM}_i.\mathsf{Setup}(1^\lambda, N) \to (\mathbf{\Gamma}_i)$: 1. Output $\mathsf{ck}_i := \mathbf{\Gamma}_{\bar{i}} \xleftarrow{\$} \mathbb{G}_{\bar{i}}^N$.	$\mathsf{CM}_i.\mathsf{Commit}(\mathsf{ck}_i, \boldsymbol{X}) \to C_{\boldsymbol{X}}$: 1. Parse ck_i as $\mathbf{\Gamma}_{\bar{i}}$. 2. If $i = 1$: output $\langle \boldsymbol{X}, \mathbf{\Gamma}_2 \rangle_e$. 3. If $i = 2$: output $\langle \mathbf{\Gamma}_1, \boldsymbol{X} \rangle_e$.

Both the Pedersen and AFGHO commitments are doubly homomorphic. Additionally, the Pedersen commitment scheme and the AFGHO commitment scheme are secure under the SXDH assumption ([AFGHO16]).

B.2 Generalized inner product arguments

We now recall the definition of generalized inner product arguments (GIPA) [BMMTV21], beginning with the relation $\mathscr{R}_{\mathsf{GIP}}$ proven by these arguments.

Definition B.5. *The indexed relation* $\mathscr{R}_{\mathsf{GIP}}$ *is the set of triples*

$$\begin{pmatrix} \mathbb{i} \\ \mathbb{x} \\ \mathbb{w} \end{pmatrix} = \begin{pmatrix} (N, \mathsf{CM}, \circledast, \mathbf{\Gamma}) \\ D \\ (\boldsymbol{V}, \boldsymbol{X}) \end{pmatrix}$$

where $N \in \mathbb{N}$ *is a power-of-two,* $(\mathsf{CM}, \circledast)$ *is an inner product commitment with key space* $\mathcal{K}$, *message space* $\mathcal{M}$, *and* $\mathbf{\Gamma} \in \mathcal{K}$, $(\boldsymbol{V}, \boldsymbol{X}, \langle \boldsymbol{V}, \boldsymbol{X} \rangle_{\circledast}) \in \mathcal{M}$ *such that* $D = \mathsf{CM}.\mathsf{Commit}(\mathbf{\Gamma}, (\boldsymbol{V}, \boldsymbol{X}, \langle \boldsymbol{V}, \boldsymbol{X} \rangle_{\circledast}))$.

B.2.1 GIPA.Reduce

In this section we present an interactive reduction of knowledge [KP23] GIPA.Reduce that reduces an instance of $\mathscr{R}_{\mathsf{GIP}}$ into a related instance of half the length using a random verifier challenge. That is, given an index $\mathbb{i} = (2^n, \mathsf{CM}, \circledast, \mathbf{\Gamma} = (\mathbf{\Gamma}_1, \mathbf{\Gamma}_2, \Gamma_3))$, GIPA.Reduce reduces the problem of checking if a tuple $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathscr{R}_{\mathsf{GIP}}$ to the problem of checking if a tuple $(\mathbb{i}', \mathbb{x}', \mathbb{w}') \in \mathscr{R}_{\mathsf{GIP}}$, where $\mathbb{i}' = (2^{n-1}, \mathsf{CM}, \circledast, \mathbf{\Gamma}' = (\mathbf{\Gamma}'_1, \mathbf{\Gamma}'_2, \Gamma_3))$ with $\mathbf{\Gamma}' \in \mathcal{K}_1^{2^{n-1}} \times \mathcal{K}_2^{2^{n-1}} \times \mathcal{K}_3$.

We define two useful helper functions. The first, ExpandEven, takes as input a vector $\boldsymbol{A}$ of length N and outputs a vector $\boldsymbol{B}$ of length $2N$ whose $2i$-th entry contains the i-th entry of $\boldsymbol{A}$; the rest of $\boldsymbol{B}$'s entries are zero. The second, ExpandOdd, is similar, but sets the $2i-1$-th entry of $\boldsymbol{B}$ to contain the i-th entry of $\boldsymbol{A}$.

The full GIPA.Reduce construction is presented below.

GIPA.Reduce

$\langle \mathsf{P}(\mathbb{i}, \mathbb{x}, \mathbb{w}), \mathsf{V}(\mathbb{i}, \mathbb{x}) \rangle$:

Parse: $\mathbb{i} = (2^n, \mathsf{CM}, \circledast, \boldsymbol{\Gamma} = (\boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2, \Gamma_3))$, $\mathbb{x} = D$ and $\mathbb{w} = (\boldsymbol{V}, \boldsymbol{X})$.

1. $\mathcal{P}$ computes the cross-inner-products $E_+ := \langle \boldsymbol{V}_O, \boldsymbol{X}_E \rangle_{\circledast}$ and $E_- := \langle \boldsymbol{V}_E, \boldsymbol{X}_O \rangle_{\circledast}$.
2. $\mathcal{P}$ computes the cross-commitments
   (a) $D_+ \leftarrow \mathsf{CM.Commit}(\boldsymbol{\Gamma}, (\mathsf{ExpandOdd}(\boldsymbol{V}_O), \mathsf{ExpandEven}(\boldsymbol{X}_E), E_+))$,
   (b) $D_- \leftarrow \mathsf{CM.Commit}(\boldsymbol{\Gamma}, (\mathsf{ExpandOdd}(\boldsymbol{V}_E), \mathsf{ExpandEven}(\boldsymbol{X}_O), E_-))$.
3. $\mathcal{P}$ sends D_+, D_- to $\mathcal{V}$.
4. $\mathcal{V}$ samples $\alpha \xleftarrow{\$} \mathbb{F}$ and sends it to $\mathcal{P}$.
5. $\mathcal{P}$ and $\mathcal{V}$ fold commitment keys and commitment:
$$\begin{aligned} \boldsymbol{\Gamma}'_1 &:= \alpha \boldsymbol{\Gamma}_{2E} + \boldsymbol{\Gamma}_{2O}, \\ \boldsymbol{\Gamma}'_2 &:= \alpha^{-1} \boldsymbol{\Gamma}_{2E} + \boldsymbol{\Gamma}_{2O}, \\ D' &:= D + \alpha D_+ + \alpha^{-1} D_-. \end{aligned}$$
6. $\mathcal{P}$ folds the witness vectors:
$$\begin{aligned} \boldsymbol{V}' &:= \alpha^{-1} \boldsymbol{v}_E + \boldsymbol{v}_O, \\ \boldsymbol{X}' &:= \alpha \boldsymbol{X}_E + \boldsymbol{X}_O. \end{aligned}$$
7. Define $\mathbb{i}' := (2^{n-1}, \mathsf{CM.Commit}, \circledast, \boldsymbol{\Gamma}' := (\boldsymbol{\Gamma}'_1, \boldsymbol{\Gamma}'_2, \Gamma_3))$, $\mathbb{x}' := D'$ and $\mathbb{w}' := (\boldsymbol{V}', \boldsymbol{X}')$.
8. $\mathcal{P}$ receives output $(\mathbb{i}', \mathbb{x}', \mathbb{w}')$ and $\mathcal{V}$ receives output $(\mathbb{i}', \mathbb{x}')$.

Lemma B.6 ([BMMTV21, Theorem 1]). *Let $\mathbb{i} = (2^n, \mathsf{CM}, \circledast, \boldsymbol{\Gamma} = (\boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2, \Gamma_3))$ be an index for $\mathscr{R}_{\mathsf{GIP}}$. Then the following is an interactive argument of knowledge for proving that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathscr{R}_{\mathsf{GIP}}$:*

1. *Prover and verifier invoke* GIPA.Reduce*: $(\mathbb{i}', \mathbb{x}', \mathbb{w}') \leftarrow \mathsf{GIPA.Reduce}(\langle \mathsf{P}(\mathbb{i}, \mathbb{x}, \mathbb{w}), \mathsf{V}(\mathbb{i}, \mathbb{x}) \rangle)$.*
2. *Prover sends $\mathbb{w}'$ to verifier.*
3. *Verifier accepts if and only if $(\mathbb{i}', \mathbb{x}', \mathbb{w}') \in \mathscr{R}_{\mathsf{GIP}}$.*

MSB folding security implies LSB folding security. The foregoing reduction deviates from the one in [BMMTV21] as it folds the even and odd parts of vectors together as $\boldsymbol{A}' := \alpha \boldsymbol{A}_E + \boldsymbol{A}_O$ (LSB folding), instead of the left and right parts ($\boldsymbol{A}' := \alpha \boldsymbol{A}_L + \boldsymbol{A}_R$ (MSB folding)). We use LSB folding because it leads to more efficient read-write streaming prover algorithms that avoid intermediate streams, and because it simplifies exposition. This choice does not affect completeness or knowledge-soundness; completeness follows from inspection, while knowledge-soundness follows from the following lemma implicit in the work of Block et al. [BHRRS20]:

Lemma B.7. *Given a polynomial-time extractor that extracts a witness $\boldsymbol{X}$ for statements of the form $((2^n, \boldsymbol{\Gamma}), D, (\boldsymbol{V}, \boldsymbol{X}))$, we can construct a polynomial-time extractor that extracts a witness for statements of the form $((2^n, \pi(\boldsymbol{\Gamma})), D, (\pi(\boldsymbol{V}), \pi(\boldsymbol{X})))$, where π is an efficiently computable and invertible permutation.*

Knowledge-soundness follows by fixing π to be the following permutation:

$$\pi(X_i) := \begin{cases} X_{2(i-1)+1} & \text{if } i \leq N/2, \\ X_{2(i-(N/2))} & \text{if } i > N/2. \end{cases}$$

B.2.2 Read-write streaming prover for GIPA.Reduce

In this section we present a streaming prover for GIPA.Reduce. We rely on the fact that the algorithms CM.Commit, ExpandOdd, and ExpandEven have efficient read-write streaming versions; this is true for all relevant instantiations of CM.Commit, since the latter is usually an inner product between the commitment key and the message. This in turn implies that Steps 2a and 2b of GIPA.Reduce can be performed in a read-write streaming manner; we call the resulting algorithm CrossCommit. We generalize the read-write streaming algorithm InnerProd to $\mathsf{InnerProd}_{\circledast}$ that computes an arbitrary inner product $\langle \cdot, \cdot \rangle_{\circledast}$ instead of the standard scalar inner product.

We are now ready to present our read-write streaming prover for GIPA.Reduce.

Read-write Streaming Algorithm 14: Prover for GIPA.REDUCE

$\mathsf{P}(\mathbb{i}, \mathbb{x}, \mathbb{w})$:

Parse: $\mathbb{i} = (2^n, \mathsf{CM}, \circledast, (\mathbf{R}_{\Gamma 1}(\mathbf{\Gamma}_1), \mathbf{R}_{\Gamma 2}(\mathbf{\Gamma}_2), \Gamma_3))$, $\mathbb{x} = D$ and $\mathbb{w} = (\mathbf{R}_V(\boldsymbol{V}), \mathbf{R}_X(\boldsymbol{X}))$.

1. Split commitment keys and witness vectors:
   - $(\mathbf{R}_{\Gamma 1E}, \mathbf{R}_{\Gamma 1O}) \leftarrow \mathsf{SplitEO}(\mathbf{R}_{\Gamma 1})$,
   - $(\mathbf{R}_{\Gamma 2E}, \mathbf{R}_{\Gamma 2O}) \leftarrow \mathsf{SplitEO}(\mathbf{R}_{\Gamma 2})$,
   - $(\mathbf{R}_{VE}, \mathbf{R}_{VO}) \leftarrow \mathsf{SplitEO}(\mathbf{R}_V)$, and
   - $(\mathbf{R}_{XE}, \mathbf{R}_{XO}) \leftarrow \mathsf{SplitEO}(\mathbf{R}_X)$.
2. Compute $E_+ \leftarrow \mathsf{InnerProd}_{\circledast}(\mathbf{R}_{VO}, \mathbf{R}_{XE})$ and $E_- \leftarrow \mathsf{InnerProd}_{\circledast}(\mathbf{R}_{VE}, \mathbf{R}_{XO})$.
3. Compute cross commitments:
   - $D_+ \leftarrow \mathsf{CrossCommit}((\mathbf{R}_{\Gamma 1}, \mathbf{R}_{\Gamma 2}, \Gamma_3), \mathbf{R}_{VO}, \mathbf{R}_{XE}, E_+)$, and
   - $D_- \leftarrow \mathsf{CrossCommit}((\mathbf{R}_{\Gamma 1}, \mathbf{R}_{\Gamma 2}, \Gamma_3), \mathbf{R}_{VE}, \mathbf{R}_{XO}, E_-)$.
4. Send D_+, D_- to $\mathcal{V}$.
5. Receive $\alpha \xleftarrow{\$} \mathbb{F}$ from $\mathcal{V}$.
6. Fold the commitment keys: $\mathsf{kFoldInPlace}(\alpha, 1, \mathbf{R}_{\Gamma 1})$ and $\mathsf{kFoldInPlace}(\alpha^{-1}, 1, \mathbf{R}_{\Gamma 2})$.
7. Fold the vectors: $\mathsf{kFoldInPlace}(\alpha^{-1}, 1, \mathbf{R}_V)$ and $\mathsf{kFoldInPlace}(\alpha, 1, \mathbf{R}_X)$.
8. Set $D' := D + \alpha D_+ + \alpha^{-1} D_-$.
9. Define $\mathbb{i}' := (2^{n-1}, \mathsf{CM}, \circledast, (\mathbf{R}_{\Gamma 1}, \mathbf{R}_{\Gamma 2}, \Gamma_3))$, $\mathbb{x}' := D'$ and $\mathbb{w}' := (\mathbf{R}_V, \mathbf{R}_X)$.
10. Store $(\mathbb{i}', \mathbb{x}', \mathbb{w}')$ for future invocations.

Lemma B.8. GIPA.Reduce *and Algorithm 14 together define a read-write streaming reduction of knowledge with the following properties:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(N)$	$O(N)$	$O(\log N)$	$O(N)$	$O(N)$	$O(1)$

Proof. The proof is similar to that of Lemma 6.1. □

B.2.3 The full GIPA protocol

We now present the full argument for $\mathscr{R}_{\mathsf{GIP}}$ that applies GIPA.Reduce iteratively to shrink the size of the instance to 1, and then directly checks this instance.

GIPA

$\langle \mathsf{P}(\mathbb{i}, \mathbb{x}, \mathbb{w}), \mathsf{V}(\mathbb{i}, \mathbb{x}) \rangle$:

Parse: $\mathbb{i} = (2^n, \mathsf{CM}, \circledast, \boldsymbol{\Gamma} = (\boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2, \Gamma_3))$, $\mathbb{x} = D$ and $\mathbb{w} = (\boldsymbol{V}, \boldsymbol{X})$.

1. Define $\mathbb{i}_1 := \mathbb{i}$, $\mathbb{x}_1 := \mathbb{x}$ and $\mathbb{w}_1 := \mathbb{w}$.
2. For i in $[1, \ldots, n]$:
3. $\quad (\mathbb{i}_{i+1}, \mathbb{x}_{i+1}, \mathbb{w}_{i+1}) \leftarrow \mathsf{GIPA.Reduce}(\langle \mathsf{P}(\mathbb{i}_i, \mathbb{x}_i, \mathbb{w}_i), \mathsf{V}(\mathbb{i}_i, \mathbb{x}_i) \rangle)$.
4. Parse $\mathbb{i}_{n+1} = (1, \mathsf{CM.Commit}, \circledast, \Gamma' = (\Gamma'_1, \Gamma'_2, \Gamma'_3))$, $\mathbb{x}_{n+1} = D'$ and $\mathbb{w}_{n+1} = (V', X')$.
5. $\mathcal{P}$ sends (V', X') to $\mathcal{V}$.
6. $\mathcal{V}$ accepts if $D' = \mathsf{CM.Commit}((\Gamma'_1, \Gamma'_2, \Gamma'_3), (V', X', \langle V', X' \rangle_{\circledast}))$.

Theorem B.9. GIPA *is a read-write streaming interactive argument of knowledge for* $\mathscr{R}_{\mathsf{GIP}}$ *with the following properties:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(N)$	$O(N)$	$O(\log N)$	$O(N)$	$O(N)$	$O(\log N)$

Proof. Completeness and knowledge soundness follow from [BMMTV21, Theorem 1]. The existence and efficiency of a read-write streaming prover for GIPA follows from Lemma B.8. □

B.3 The MIPP Protocol

A key building block of the polynomial commitment schemes from [BMMTV21] and [Lee21] is an interactive argument for the MIPP relation from [BMMTV21], which allows a prover to convince a verifier that the multiscalar multiplication $\langle \boldsymbol{v}, \boldsymbol{X} \rangle_{\mathbb{G}} = E$, where $\boldsymbol{X} \in \mathbb{G}^N$ is a private group vector committed to in the AFGHO commitment D and $\boldsymbol{v} \in \mathbb{F}^N$ is a public field vector shared by both the prover and verifier.

Formally, the Multi-exponentiation Inner Product (MIPP) relation $\mathscr{R}_{\mathsf{MIPP}}$ is defined as follows:

Definition B.10 (MIPP relation)**.** *The indexed relation* $\mathscr{R}_{\mathsf{MIPP}}$ *is the set of triples*

$$\begin{pmatrix} \mathbb{i}, \\ \mathbb{x}, \\ \mathbb{w} \end{pmatrix} = \begin{pmatrix} (N, \boldsymbol{\Gamma}_2), \\ (D, \boldsymbol{v}, E), \\ \boldsymbol{X} \end{pmatrix}$$

where N *is an integer,* $\boldsymbol{\Gamma}_2 \in \mathbb{G}_2^N$, $D \in \mathbb{G}_T$, $\boldsymbol{v} \in \mathbb{F}^N$, $E \in \mathbb{G}_1$ *and* $\boldsymbol{X} \in \mathbb{G}_1^N$ *such that*

$$D = \mathsf{CM}_1(\boldsymbol{\Gamma}_2, \boldsymbol{X}), \quad E = \langle \boldsymbol{v}, \boldsymbol{X} \rangle_{\mathbb{G}}$$

Notice that $\mathscr{R}_{\mathsf{MIPP}}$ is a specific instantiation of $\mathscr{R}_{\mathsf{GIP}}$, with $\circledast : \mathbb{F} \times \mathbb{G}_1 \to \mathbb{G}_1$ defined to be $a \circledast B = a \cdot B$ and the inner product commitment $\mathsf{CM}_{\mathsf{MIPP}}.\mathsf{Commit}$, given commitment key $\boldsymbol{\Gamma}_2 \in \mathbb{G}_2^N$, defined to be:

$$\mathsf{CM}_{\mathsf{MIPP}}.\mathsf{Commit}((\perp, \boldsymbol{\Gamma}_2, \perp), (\boldsymbol{a}, \boldsymbol{B}, E)) = (\mathsf{CM}_1.\mathsf{Commit}(\boldsymbol{\Gamma}_2, \boldsymbol{a}, \boldsymbol{B}), E).$$

The following lemma thus follows naturally from Theorem B.9.

Lemma B.11 (Corollary of Theorem B.9)**.** MIPP *is a read-write streaming interactive argument of knowledge for* $\mathscr{R}_{\mathsf{MIPP}}$ *with the following properties:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(N)$	$O(N)$	$O(\log N)$	$O(N)$	$O(N)$	$O(\log N)$

Sublinear verification. Bunz et al. [BMMTV21] describe an optimization that reduces the GIPA verifier's time from $O(N)$ to $O(\log N)$ by relying on a structured commitment key. Roughly, in the GIPA protocol, the verifier's work can be rewritten so that at each round, instead of folding the commitment keys, the verifier only needs to do a small amount of work to check consistency of the folded commitments; then, at the end of the protocol, the verifier needs to check the opening of a single N-sized commitment derived form its challenges. Bunz et al. show that this commitment can be viewed as a *polynomial* commitment to a structured polynomial, and leverage the KZG polynomial commitment scheme to allow the prover to *prove* correct commitment opening. The prover's work here consists of generating, committing to, and opening this polynomial commitment, and all steps can be done in a read-write streaming manner (and some even in a read-only streaming manner [BCHO22]).

For the overall MIPP verifier to run in $O(\log N)$ time, in addition to the foregoing GIPA optimization, the verifier also needs to be able to succinctly operate over the vector $\boldsymbol{v}$. This is indeed the case in the applications we consider, namely polynomial evaluation, where $\boldsymbol{v}$ is derived from an evaluation point for a multilinear polynomial.

B.4 The FIP protocol

In this section, we present another important building block of the polynomial commitment schemes from [BMMTV21] and [Lee21], the FIP protocol, which at a high level allows a prover to convince a verifier of the multiscalar multiplication $\langle \boldsymbol{v}, \boldsymbol{y} \rangle_{\mathbb{G}} = e$, where $\boldsymbol{y} \in \mathbb{F}^N$ is a private field vector committed to in the Pedersen commitment D and $\boldsymbol{v} \in \mathbb{F}^N$ is a public field vector shared by both the prover and verifier.

Formally, the Field Inner Product (FIP) relation $\mathscr{R}_{\mathsf{FIP}}$ is defined analogously to $\mathscr{R}_{\mathsf{MIPP}}$ as follows:

Definition B.12 (FIP relation)**.** *The indexed relation $\mathscr{R}_{\mathsf{FIP}}$ is the set of triples*

$$\begin{pmatrix} \mathbb{i}, \\ \mathbb{x}, \\ \mathbb{w} \end{pmatrix} = \begin{pmatrix} (N, \boldsymbol{\Gamma}_1), \\ (D, \boldsymbol{v}, e), \\ \boldsymbol{y} \end{pmatrix}$$

where N is an integer, $\boldsymbol{\Gamma}_1 \in \mathbb{G}_1^N$, $D \in \mathbb{G}_1$, $\boldsymbol{v} \in \mathbb{F}^N$, $e \in \mathbb{F}$ and $\boldsymbol{y} \in \mathbb{F}^N$ such that

$$D = \mathsf{CM}_P(\boldsymbol{\Gamma}_1, \boldsymbol{y}), \quad e = \langle \boldsymbol{v}, \boldsymbol{y} \rangle_{\mathbb{F}}$$

Notice that $\mathscr{R}_{\mathsf{FIP}}$ is a specific instantiation of $\mathscr{R}_{\mathsf{GIP}}$, with $\circledast : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ defined to be $a \circledast b = a \cdot b$ and the inner product commitment $\mathsf{CM}_{\mathsf{FIP}}.\mathsf{Commit}$, given commitment key $\boldsymbol{\Gamma}_1 \in \mathbb{G}_1^N$, defined to be:

$$\mathsf{CM}_{\mathsf{FIP}}.\mathsf{Commit}((\boldsymbol{\Gamma}_1, \perp, \perp), (\boldsymbol{a}, \boldsymbol{b}, e)) = (\mathsf{CM}_P.\mathsf{Commit}(\boldsymbol{\Gamma}_1, \boldsymbol{b}), \boldsymbol{a}, E).$$

The following lemma thus follows naturally from Theorem B.9.

Theorem B.13 (Corollary of Theorem B.9)**.** FIP *is a read-write streaming interactive argument of knowledge for $\mathscr{R}_{\mathsf{FIP}}$ with the following properties:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(N)$	$O(N)$	$O(\log N)$	$O(N)$	$O(N)$	$O(\log N)$

A similar verifier efficiency optimization as in the MIPP protocol can be applied to the FIP protocol; we refer the reader to Bünz et al. [BMMTV21] for details.

C Constructing polynomial commitment schemes with square-root SRS

Wahby et al. [WTSTW18] proposed a novel recipe for constructing polynomial commitment schemes using inner product arguments, where the size of the commitment key is sublinear in the size of the polynomial being committed to. This blueprint is also used in the PC scheme of Bünz et al. [BMMTV21] and Dory [Lee21]. The blueprint proceeds by viewing an n-variate multilinear polynomial $p(\boldsymbol{X})$, represented by its evaluation over the boolean hypercube $\boldsymbol{p}$, as a matrix $\mathbf{M} \in \mathbb{F}^{\sqrt{N} \times \sqrt{N}}$ defined as follows:

$$M_{ij} := p_k \text{ for } k = i \cdot 2^m + j.$$

where $N := 2^n$ and $m := n/2$. Without loss of generality, we can assume that for the n-variate multilinear polynomials we consider, n is even. This is because if n is odd, we can just add a 'dummy' variable that is not included in the polynomial, incurring a cost overhead of at most a multiplicative factor of 2.[22]

Their key observation is that evaluating $p(X_1, \ldots, X_n)$ at a point $\boldsymbol{z} = (z_1, z_2, \ldots, z_n)$ is equivalent to computing the vector-matrix-vector (VMV) product $\boldsymbol{\ell}^\top \mathbf{M} \boldsymbol{r}$, where $\boldsymbol{\ell} := [\mathrm{eq}(\boldsymbol{z}_L, \mathsf{bin}_n(i))]_{i=0}^{\sqrt{N}-1} = \otimes_{i=1}^{m}(1 - z_i, z_i)$ and $\boldsymbol{r} := [\mathrm{eq}(\boldsymbol{z}_R, \mathsf{bin}_n(i))]_{i=0}^{\sqrt{N}-1} = \otimes_{i=m+1}^{n}(1 - z_i, z_i)$. This can be done via 'vector-matrix-vector product' arguments.

VMV arguments. A vector-matrix-vector product argument is a tuple of algorithms $\mathsf{VMV} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Eval})$ with the following syntax.

- $\mathsf{VMV.Setup}(1^\lambda, N) \to (\mathsf{ck}, \mathsf{vk})$. On input a security parameter λ (in unary), and a maximum dimension bound $N \in \mathbb{N}$, VMV.Setup samples committer and verifier keys ck and vk.
- $\mathsf{VMV.Commit}(\mathsf{ck}, \mathbf{M}) \to C_{\mathbf{M}}$. On input ck and a matrix $\mathbf{M} \in \mathbb{F}^{\sqrt{N} \times \sqrt{N}}$, VMV.Commit outputs a commitment $C_{\mathbf{M}}$ to $\mathbf{M}$.
- $\mathsf{VMV.Eval}(\langle \mathcal{P}(\mathsf{ck}, (C_{\mathbf{M}}, \boldsymbol{\ell}, \boldsymbol{r}, y), \mathbf{M}), \mathcal{V}(\mathsf{vk}, (C_{\mathbf{M}}, \boldsymbol{\ell}, \boldsymbol{r}, y)) \rangle)$. $\mathcal{P}$ and $\mathcal{V}$ engage in an interactive protocol that convinces $\mathcal{V}$ that $C_{\mathbf{M}}$ commits to a matrix $\mathbf{M}$ satisfying the claim "$\boldsymbol{\ell}^\top \cdot \mathbf{M} \cdot \boldsymbol{r} = y$".

A vector-matrix-vector product argument VMV must satisfy completeness and extractability properties analogous to those of a polynomial commitment scheme. We show how to use a VMV argument to obtain a PC scheme in the next section, but first we make a comment about our presentation for the rest of the paper.

Presentation. For the subsequent polynomial commitment schemes, for ease of exposition we present the evaluation protocol as an interactive protocol PC.Eval. The standard non-interactive notions PC.Open and PC.Check correspond to the prover and verifier respectively of the non-interactive version of PC.Eval obtained via the Fiat-Shamir transform [FS86].

C.1 Constructing VMV arguments

As a prerequisite, we define the following instantiations of $\mathscr{R}_{\mathsf{GIP}}$ and their associated arguments:

- $\mathscr{R}_{\mathsf{in}}$ is $\mathscr{R}_{\mathsf{GIP}}$ specialized to a commitment scheme $\mathsf{CM}_{\mathsf{in}}$ with message space $\mathbb{F}^{\sqrt{N}}$ and commitment space $\mathbb{G}_1$, and inner product map $\circledast : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ defined to be $a \circledast b = a \cdot b$. $\mathsf{Arg}_{\mathsf{in}}$ is an interactive argument for $\mathscr{R}_{\mathsf{in}}$.
- $\mathscr{R}_{\mathsf{out}}$ is $\mathscr{R}_{\mathsf{GIP}}$ specialized to a commitment scheme $\mathsf{CM}_{\mathsf{out}}$ with message space $\mathbb{G}_1^{\sqrt{N}}$, and an inner product map $\star : \mathbb{F} \times \mathbb{G} \to \mathbb{G}$ defined to be $a \star B = a \cdot B$. $\mathsf{Arg}_{\mathsf{out}}$ is an interactive argument for $\mathscr{R}_{\mathsf{out}}$.

[22] We can in fact handle an odd-number of variables without incurring this overhead by using a non-square matrix, but we omit that discussion for ease of exposition.

The blueprint is as follows: commit to a matrix $\mathbf{M} \in \mathbb{F}^{\sqrt{N} \times \sqrt{N}}$ by first commiting to the rows using $\mathsf{CM}_{\mathsf{in}}$ to obtain $C_i \leftarrow \mathsf{CM}_{\mathsf{in}}.\mathsf{Commit}(\mathsf{ck}_{\mathsf{in}}, \boldsymbol{M}_i)$ for each $i \in [0, 1, \ldots, \sqrt{N} - 1]$, and then committing to $\boldsymbol{C} := [C_i]_{i=0}^{\sqrt{N}-1}$ using $\mathsf{CM}_{\mathsf{out}}$ to obtain $C_{\mathbf{M}} \leftarrow \mathsf{CM}_{\mathsf{out}}.\mathsf{Commit}(\mathsf{ck}_{\mathsf{out}}, \boldsymbol{C})$.

To show that the VMV product $\boldsymbol{\ell}^\top \mathbf{M} \boldsymbol{r} = y$ for the matrix $\mathbf{M}$ committed to in $C_{\mathbf{M}}$, the prover and verifier engage in an interactive argument $\mathsf{Arg}_{\mathsf{out}}(\langle \mathcal{P}(\mathsf{ck}_{\mathsf{out}}, (C_{\mathbf{M}}, \boldsymbol{\ell}, C'), \boldsymbol{C}), \mathcal{V}(\mathsf{vk}_{\mathsf{out}}, (C_{\mathbf{M}}, \boldsymbol{\ell}, C')) \rangle)$ to show that $\sum_{i=0}^{\sqrt{N}-1} \ell_i \cdot C_i = C'$. This implies that C' must be a commitment to $\boldsymbol{a} := \boldsymbol{\ell}^\top \mathbf{M}$ under the commitment scheme $\mathsf{CM}_{\mathsf{in}}$. The prover is left to convince the verifier that $\langle \boldsymbol{a}, \boldsymbol{r} \rangle_{\mathbb{F}} = y$, which is done using $\mathsf{Arg}_{\mathsf{in}}$. The full construction is as follows:

$\mathsf{VMV.Setup}(1^\lambda, N) \to (\mathsf{ck}, \mathsf{vk})$:
1. Sample $(\mathsf{ck}_{\mathsf{in}}, \mathsf{vk}_{\mathsf{in}}) \leftarrow \mathsf{CM}_{\mathsf{in}}.\mathsf{Setup}(1^\lambda, \sqrt{N})$.
2. Sample $(\mathsf{ck}_{\mathsf{out}}, \mathsf{vk}_{\mathsf{out}}) \leftarrow \mathsf{CM}_{\mathsf{out}}.\mathsf{Setup}(1^\lambda, \sqrt{N})$.
3. Output $(\mathsf{ck} := (\mathsf{ck}_{\mathsf{in}}, \mathsf{ck}_{\mathsf{out}}), \mathsf{vk} := (\mathsf{vk}_{\mathsf{in}}, \mathsf{vk}_{\mathsf{out}}))$

$\mathsf{VMV.Commit}(\mathsf{ck}, \mathbf{M}) \to C_{\mathbf{M}}$:
1. Parse ck as $(\mathsf{ck}_{\mathsf{in}}, \mathsf{ck}_{\mathsf{out}})$.
2. For all $i \in [0, 1, \ldots, \sqrt{N} - 1]$: Commit to the i-th row as $C_i \leftarrow \mathsf{CM}_{\mathsf{in}}.\mathsf{Commit}(\mathsf{ck}_{\mathsf{in}}, \mathbf{M}_i)$.
3. Output $C_{\mathbf{M}} \leftarrow \mathsf{CM}_{\mathsf{out}}.\mathsf{Commit}(\mathsf{ck}_{\mathsf{out}}, \boldsymbol{C})$.

$\mathsf{VMV.Eval}(\langle \mathcal{P}(\mathsf{ck}, \mathbb{x}, \mathbf{M}), \mathcal{V}(\mathsf{vk}, \mathbb{x}) \rangle)$:
Parse: $\mathsf{ck} = (\mathsf{ck}_{\mathsf{in}}, \mathsf{ck}_{\mathsf{out}})$, $\mathsf{vk} = (\mathsf{vk}_{\mathsf{in}}, \mathsf{vk}_{\mathsf{out}})$ and $\mathbb{x} = (C_{\mathbf{M}}, \boldsymbol{\ell}, \boldsymbol{r}, y)$.
1. For all $i \in [0, 1, \ldots, \sqrt{N} - 1]$: $\mathcal{P}$ computes $C_i \leftarrow \mathsf{CM}_{\mathsf{in}}.\mathsf{Commit}(\mathsf{ck}_{\mathsf{in}}, \mathbf{M}_i)$.
2. $\mathcal{P}$ sets
$$\begin{aligned} \boldsymbol{X} &:= \boldsymbol{C}, & \boldsymbol{a} &:= \boldsymbol{\ell}^\top \mathbf{M}, \\ D_1 &:= C_{\mathbf{M}}, & D_2 &:= \mathsf{CM}_{\mathsf{in}}.\mathsf{Commit}(\mathsf{ck}_{\mathsf{in}}, \boldsymbol{a}), \\ E_1 &:= \langle \boldsymbol{\ell}, \boldsymbol{X} \rangle_{\mathbb{G}}, & e_2 &:= \langle \boldsymbol{r}, \boldsymbol{a} \rangle_{\mathbb{F}}. \end{aligned}$$
3. $\mathcal{P}$ sends E_1 to $\mathcal{V}$.
4. $\mathcal{V}$ sets $D_1 := C_{\mathbf{M}}$, $D_2 := E_1$ and $e_2 := y$.
5. Set $\mathbb{i}_1 := (\sqrt{N}, \mathsf{ck}_{\mathsf{out}})$, $\mathbb{x}_1 := (D_1, \boldsymbol{\ell}, E_1)$, $\mathbb{i}_2 := (\sqrt{N}, \mathsf{ck}_{\mathsf{in}})$ and $\mathbb{x}_2 := (D_2, \boldsymbol{r}, e_2)$.
6. $\mathcal{V}$ accepts if: $\langle \mathcal{P}_{\mathsf{out}}(\mathbb{i}_1, \mathbb{x}_1, \boldsymbol{X}), \mathcal{V}_{\mathsf{out}}(\mathbb{i}_1, \mathbb{x}_1) \rangle = 1$ and $\langle \mathcal{P}_{\mathsf{in}}(\mathbb{i}_2, \mathbb{x}_2, \boldsymbol{a}), \mathcal{V}_{\mathsf{in}}(\mathbb{i}_2, \mathbb{x}_2) \rangle = 1$.

Clearly $\mathsf{VMV.Commit}$ has a read-write streaming algorithm if $\mathsf{CM}_{\mathsf{in}}.\mathsf{Commit}$ and $\mathsf{CM}_{\mathsf{out}}.\mathsf{Commit}$ have read-write streaming algorithms; this is true for all known instantiations (see Appendices C.3 and C.4). We are hence left to show that $\mathsf{VMV.Eval}$ has a read-write streaming prover. Our algorithm for the latter relies on the following helper functions, all of which require $O(\log N)$ random access space:

- $\mathsf{ExtractRow}$, given as input a read-stream for a matrix specified in row-major order, outputs the rows of a matrix one at a time;
- $\mathsf{CommitMatrix}$ computes commitments to these rows;
- VMProd computes vector matrix products of the form $\boldsymbol{\ell}^\top \mathbf{M}$; and
- $\mathsf{TensorProd}$, which compute tensor products of the form $\boldsymbol{c} := \otimes_{i=1}^{n} (a_i, b_i)$.

$\mathsf{ExtractRow}(N, \mathbf{R}_M) \to \mathbf{S}_R$:
1. For i in $[1, \ldots, \sqrt{N}]$, emit $m_i \leftarrow \mathbf{R}_M.\mathsf{read}()$.

$\mathsf{CommitMatrix}(N, \mathbf{R}_{\mathsf{ck}}, \mathbf{R}_M) \to \mathbf{S}_C$:
1. For i in $[1, \ldots, \sqrt{N}]$, emit
$D \rightsquigarrow \mathsf{CM}_{\mathsf{in}}.\mathsf{Commit}(\mathbf{R}_{\mathsf{ck}}) \rightsquigarrow \mathsf{ExtractRow}(N, \mathbf{R}_M)$.

$\mathsf{VMProd}(N, \mathbf{R}_\ell, \mathbf{R}_M) \to \mathbf{W}_S$:
1. Initialize the read stream $\mathbf{R}_S([0]_{i=0}^{\sqrt{N}-1})$.
2. For i in $[0, \ldots, \sqrt{N}-1]$:
3. $\quad \ell \leftarrow \mathbf{R}_\ell.\mathsf{read}()$.
4. $\quad \mathbf{R}_R \leftarrow \mathsf{ExtractRow}(N, \mathbf{R}_M)$.
5. $\quad \mathbf{R}_S \leftarrow \mathsf{LinComb}(1, \ell, \mathbf{R}_S, \mathbf{R}_R)$.
6. Output $\mathbf{W}_S \leftarrow \mathbf{R}_S.\mathsf{swapmode}()$.

$\mathsf{TensorProd}(n, \mathbf{R}_a, \mathbf{R}_b) \to \mathbf{W}_c$:
1. Initialize the read stream $\mathbf{R}_c(1)$.
2. For i in $[1, \ldots, n]$:
3. $\quad a \leftarrow \mathbf{R}_a.\mathsf{read}()$, $b \leftarrow \mathbf{R}_b.\mathsf{read}()$.
4. $\quad \mathbf{R}_{ia} \rightsquigarrow \mathsf{Map}(x \mapsto a \cdot x) \rightsquigarrow \mathbf{R}_c$.
5. $\quad \mathbf{R}_{ib} \rightsquigarrow \mathsf{Map}(x \mapsto b \cdot x) \rightsquigarrow \mathbf{R}_c$.
6. $\quad$ Set $\mathbf{R}_c \leftarrow \mathsf{Concat}(\mathbf{R}_{ia}, \mathbf{R}_{ib})$.
7. Output $\mathbf{W}_c \leftarrow \mathbf{R}_c.\mathsf{swapmode}()$.

Read-write streaming prover for VMV.Eval. We now present the full construction of a read-write streaming prover for VMV.Eval. We assume $\mathsf{Arg}_{\mathsf{in}}$ and $\mathsf{Arg}_{\mathsf{out}}$ all have read-write streaming provers, which is the case for all our instantiations (see Appendices C.3 and C.4).

Read-write Streaming Algorithm 15: prover for VMV.EVAL

$\mathsf{P}(\mathsf{ck}, \mathbb{x}, \mathbf{R}_M.\mathsf{init}(\mathbf{M}))$:
Parse: $\mathsf{ck} = (\mathbf{R}_{\mathsf{ckIn}}.\mathsf{init}(\mathsf{ck}_{\mathsf{in}}), \mathbf{R}_{\mathsf{ckOut}}.\mathsf{init}(\mathsf{ck}_{\mathsf{out}}))$ and $\mathbb{x} = (C_{\mathbf{M}}, \mathbf{R}_\ell.\mathsf{init}(\boldsymbol{\ell}), \mathbf{R}_r.\mathsf{init}(\boldsymbol{r}), y)$.
1. Compute commitments to the rows of $\mathbf{M}$: $\mathbf{R}_X \leftarrow \mathsf{CommitMatrix}(N, \mathbf{R}_{\mathsf{ckIn}}, \mathbf{R}_M)$.
2. Compute $\boldsymbol{\ell}^\top \mathbf{M}$: $\mathbf{R}_a \leftarrow \mathsf{VMProd}(N, \mathbf{R}_\ell, \mathbf{R}_M)$.
3. Compute $D_2 \leftarrow \mathsf{CM}_{\mathsf{in}}.\mathsf{Commit}(\mathbf{R}_{\mathsf{ckIn}}, \mathbf{R}_a)$.
4. Compute $E_1 \leftarrow \mathsf{InnerProd}(\mathbf{R}_\ell, \mathbf{R}_X)$.
5. Compute $e_2 \leftarrow \mathsf{InnerProd}(\mathbf{R}_r, \mathbf{R}_a)$.
6. Send E_1 to $\mathcal{V}$.
7. Set $\mathbb{i}_1 := (\sqrt{N}, \mathbf{R}_{\mathsf{ckOut}})$, $\mathbb{x}_1 := (C_{\mathbf{M}}, \mathbf{R}_\ell, E_1)$, $\mathbb{i}_2 := (\sqrt{N}, \mathbf{R}_{\mathsf{ckIn}})$, $\mathbb{x}_2 := (D_2, \mathbf{R}_\ell, e_2)$.
8. Run: $\mathsf{Arg}_{\mathsf{out}}(\langle \mathcal{P}(\mathbb{i}_1, \mathbb{x}_1, \mathbf{R}_X), \mathcal{V}(\mathbb{i}_1, \mathbb{x}_1)\rangle)$ and $\mathsf{Arg}_{\mathsf{in}}(\langle \mathcal{P}(\mathbb{i}_2, \mathbb{x}_2, \mathbf{R}_a), \mathcal{V}(\mathbb{i}_2, \mathbb{x}_2)\rangle)$.

C.2 Constructing PC schemes from VMV arguments

Given a vector-matrix-vector product argument VMV, the blueprint to construct a polynomial commitment scheme PC for n-variate multilinear polynomials is as follows.

$\mathsf{PC.Setup}(1^\lambda, n) \to (\mathsf{ck}, \mathsf{vk})$:
1. Define $N := 2^n$.
2. Output $(\mathsf{ck}, \mathsf{vk}) \leftarrow \mathsf{VMV.Setup}(1^\lambda, N)$.

$\mathsf{PC.Commit}(\mathsf{ck}, \boldsymbol{p}) \to C_{\mathbf{M}}$:
1. Define $\mathbf{M} \in \mathbb{F}^{\sqrt{N} \times \sqrt{N}}$ with $M_{ij} := p_{i \cdot \sqrt{N} + j}$.
2. Output $C_{\mathbf{M}} \leftarrow \mathsf{VMV.Commit}(\mathsf{ck}, \mathbf{M})$.

$\mathsf{PC.Eval}(\langle \mathcal{P}(\mathsf{ck}, \mathbb{x}, \boldsymbol{p}), \mathcal{V}(\mathsf{vk}, \mathbb{x})\rangle)$:
Parse: $\mathbb{x} = (C_{\mathbf{M}}, \boldsymbol{z}, y)$.
1. $\mathcal{P}$ defines $\mathbf{M}$ with $M_{ij} := p_{i \cdot \sqrt{N} + j}$.
2. $\mathcal{P}$ and $\mathcal{V}$ set $\boldsymbol{\ell} := \otimes_{i=1}^{n/2}(1 - z_i, z_i)$ and $\boldsymbol{r} := \otimes_{i=n/2+1}^{n}(1 - z_i, z_i)$.
3. Define $\mathbb{x}_{\mathsf{VMV}} := (C_{\mathbf{M}}, \boldsymbol{\ell}, \boldsymbol{r}, y)$ and $\mathbb{w}_{\mathsf{VMV}} := \mathbf{M}$.
4. Output $\langle \mathcal{P}_{\mathsf{VMV}}(\mathsf{ck}, \mathbb{x}_{\mathsf{VMV}}, \mathbb{w}_{\mathsf{VMV}}), \mathcal{V}_{\mathsf{VMV}}(\mathsf{vk}, \mathbb{x}_{\mathsf{VMV}})\rangle$.

Since all the helper functions run in $O(N)$ time, the foregoing PC scheme inherits the efficiency and RW streaming prover of the underlying VMV argument.

C.3 The Hyrax PC scheme

The Hyrax PC scheme from [WTSTW18] is obtained via the aforementioned blueprint, with $\mathsf{CM}_{\mathsf{in}}$ and $\mathsf{Arg}_{\mathsf{in}}$ corresponding to FIP, and $\mathsf{CM}_{\mathsf{out}}.\mathsf{Commit}(\perp, \boldsymbol{X}) := \boldsymbol{X}$ being the identity commitment, and $\mathsf{Arg}_{\mathsf{out}}$ being the naive argument, where the verifier directly checks that $\sum_{i=0}^{N-1} \ell_i \cdot X_i = X'$ in $O(\sqrt{N})$ time. The following lemma follows from [WTSTW18, Lemma 5] and the fact that all of the aforementioned schemes have read-write streaming provers.

Lemma C.1. Hyrax *is a read-write streaming polynomial commitment scheme for n-variate multilinear polynomials with the following efficiency:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(\sqrt{N})$	$O(N)$	$O(\log N)$	$O(N)$	$O(\sqrt{N})$	$O(\log N)$

C.4 The PC scheme implicit in BMMTV21

Bünz et al. [BMMTV21] construct a univariate PC scheme that achieves square-root SRS size, linear prover time, and logarithmic verifier time. Their scheme can be adapted to the multilinear setting without affecting performance by adapting the VMV argument implicit in their scheme. Their VMV scheme is obtained via the aforementioned blueprint, with $\mathsf{CM}_{\mathsf{in}}$ and $\mathsf{Arg}_{\mathsf{in}}$ corresponding to FIP, and $\mathsf{CM}_{\mathsf{out}}$ and $\mathsf{Arg}_{\mathsf{out}}$ corresponding to MIPP. As an optimization, the scheme uses a structured commitment key to obtain a sublinear verifier as explained in Appendices B.3 and B.4. The following lemma follows from the fact that all of the aforementioned components have read-write streaming provers.

Lemma C.2. *The PC scheme implicit in [BMMTV21] is a read-write streaming polynomial commitment scheme for n-variate multilinear polynomials with the following efficiency:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(\sqrt{N})$	$O(N)$	$O(\log N)$	$O(N)$	$O(\log N)$	$O(\log N)$

D Vector-matrix-vector product arguments from Dory

In this section we describe the VMV argument from Dory [Lee21], which departs slightly from the blueprint presented in Appendix C.1, by essentially running a MIPP and FIP in parallel with modified versions MIPP.Reduce and FIP.Reduce to allow the commitment keys for each round to be fixed a priori (and not obtained by folding the commitment key from the prior round).

This allows the verifier to know the commitment key for the final round without having to fold it itself, thus avoiding the need to have either an $O(N)$ verification time overhead or the need to have the prover provide a proof of correct folding like in [BMMTV21], which would require a trusted setup. At a high level, Dory's VMV argument works as follows:[23]

Setup. As part of its commitment key, Setup samples the the unstructured commitment keys for round 1 of the FIP and MIPP protocols, $\boldsymbol{\Gamma}_{1,1} \xleftarrow{\$} \mathbb{G}_1^N$ and $\boldsymbol{\Gamma}_{2,1} \xleftarrow{\$} \mathbb{G}_2^N$. For each round $i \in [2, 3, \ldots, n]$, Setup sets the commitment keys for round i to be $\boldsymbol{\Gamma}_{1,i} := (\boldsymbol{\Gamma}_{1,i-1})_L$ and $\boldsymbol{\Gamma}_{2,i} := (\boldsymbol{\Gamma}_{2,i-1})_L$. In addition, the commitment key contains certain preprocessed information required for the modified reductions for FIP and MIPP.

Commit. Dory commits to a matrix $\mathbf{M} \in \mathbb{F}^{N \times N}$ in the standard way: by first Pedersen commiting to the rows to obtain $C_i \leftarrow \mathsf{CM}_P.\mathsf{Commit}(\boldsymbol{\Gamma}_{1,1}, \boldsymbol{M}_i)$ for each $i \in [0, 1, \ldots, N-1]$, and then AFGHO committing to $\boldsymbol{C}$ to obtain $C_{\mathbf{M}} \leftarrow \mathsf{CM}_1.\mathsf{Commit}(\boldsymbol{\Gamma}_{2,1}, \boldsymbol{C})$.

Eval. To show that the VMV product $\boldsymbol{\ell}^T \mathbf{M} \boldsymbol{r} = y$, for the matrix $\mathbf{M}$ committed to in $C_{\mathbf{M}}$, prior schemes have first used an MIPP argument to show that $\sum_{i=0}^{N-1} \ell_i \cdot C_i = C'$, where C' must be the commitment to $\boldsymbol{a} := \boldsymbol{\ell}^T \mathbf{M}$. Then, to convince the verifier that $\langle \boldsymbol{a}, \boldsymbol{r} \rangle_{\mathbb{F}} = y$, the prover would then use an FIP argument.

Dory essentially runs the MIPP and FIP in parallel but runs the FIP 'in $\mathbb{G}_2$' by lifting the vector $\boldsymbol{a} \in \mathbb{F}^N$ to $\boldsymbol{A} := \boldsymbol{a} \cdot \Gamma_{2,\mathsf{fin}} \in \mathbb{G}_2^N$ using a generator $\Gamma_{2,\mathsf{fin}} \xleftarrow{\$} \mathbb{G}_2$. This is done because by dealing with the group vectors $\boldsymbol{C} \in \mathbb{G}_1^N$ and $\boldsymbol{A} \in \mathbb{G}_2^N$, one can fold the commitment keys $\boldsymbol{\Gamma}_1$ and $\boldsymbol{\Gamma}_2$ into $\boldsymbol{C}$ and $\boldsymbol{A}$ respectively, enabling Dory to switch to a completely new set of commitment keys $\boldsymbol{\Gamma}'_1 \in \mathbb{G}_1^{N/2}$ and $\boldsymbol{\Gamma}'_2 \in \mathbb{G}_2^{N/2}$ (that do not need to depend on $\boldsymbol{\Gamma}_1$ and $\boldsymbol{\Gamma}_2$) in the every round.

Scalar pairing product. We first define the scalar pairing product relation $\mathscr{R}_{\mathsf{SPP}}$, as defined in [Lee21] and present their interactive argument Dory.InnerProd for $\mathscr{R}_{\mathsf{SPP}}$. Then in Appendix D.4 we show how to use Dory.InnerProd to construct a VMV argument.

Definition D.1 (scalar pairing product relation)**.** *The indexed relation $\mathscr{R}_{\mathsf{SPP}}$ is the set of triples*

$$\begin{pmatrix} \mathbb{i}, \\ \mathbb{x}, \\ \mathbb{w} \end{pmatrix} = \begin{pmatrix} (N, \boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2), \\ (C, D_1, D_2, E_1, E_2, \boldsymbol{\ell}, \boldsymbol{r}), \\ (\boldsymbol{X}, \boldsymbol{Y}) \end{pmatrix}$$

where N is an integer, $\boldsymbol{\Gamma}_1 \in \mathbb{G}_1^N$, $\boldsymbol{\Gamma}_2 \in \mathbb{G}_2^N$, $C, D_1, D_2 \in \mathbb{G}_T$, $E_1 \in \mathbb{G}_1$, $E_2 \in \mathbb{G}_2$, $\boldsymbol{\ell}, \boldsymbol{r} \in \mathbb{F}^N$, $\boldsymbol{X} \in \mathbb{G}_1^N$ and $\boldsymbol{Y} \in \mathbb{G}_2^N$ such that

$$\begin{aligned} &C = \langle \boldsymbol{X}, \boldsymbol{Y} \rangle_e, \\ &D_1 = \mathsf{CM}_1.\mathsf{Commit}(\boldsymbol{\Gamma}_2, \boldsymbol{X}), \quad D_2 = \mathsf{CM}_2.\mathsf{Commit}(\boldsymbol{\Gamma}_1, \boldsymbol{Y}), \\ &E_1 = \langle \boldsymbol{\ell}, \boldsymbol{X} \rangle_{\mathbb{G}}, \qquad\qquad\quad E_2 = \langle \boldsymbol{r}, \boldsymbol{Y} \rangle_{\mathbb{G}}. \end{aligned}$$

[23] For ease of exposition, we consider committing to and evaluating VMV products over matrices of dimension $N \times N$.

D.1 Dory.Reduce

In this section we present an interactive reduction of knowledge Dory.Reduce [24] that reduces an instance of $\mathscr{R}_{\mathsf{SPP}}$ into a related instance of half the length using a random verifier challenge. That is, given an index $\mathbb{i} = (2^n, \mathbf{\Gamma}_1, \mathbf{\Gamma}_2)$, and commitment keys $\mathbf{\Gamma}'_1 \xleftarrow{\$} \mathbb{G}_1^{2^{n-1}}$ and $\mathbf{\Gamma}'_2 \xleftarrow{\$} \mathbb{G}_2^{2^{n-1}}$ that are fixed a priori but randomly sampled, Dory.Reduce reduces the problem of checking if a tuple $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathscr{R}_{\mathsf{IPP}}$ to the problem of checking if $(\mathbb{i}', \mathbb{x}', \mathbb{w}') \in \mathscr{R}_{\mathsf{GIP}}$, where $\mathbb{i}' = (2^{n-1}, \mathbf{\Gamma}'_1, \mathbf{\Gamma}'_2)$.

Dory.Reduce

$\langle \mathcal{P}((\mathbf{\Gamma}'_1, \mathbf{\Gamma}'_2), \mathbb{i}, \mathbb{x}, \mathbb{w}), \mathcal{V}(\mathbb{i}, \mathbb{x}) \rangle$:
Parse: $\mathbb{i} = (2^n, \mathbf{\Gamma}_1, \mathbf{\Gamma}_2)$, $\mathbb{x} = (C, D_1, D_2, E_1, E_2, \boldsymbol{\ell}, \boldsymbol{r})$ and $\mathbb{w} = (\boldsymbol{X}, \boldsymbol{Y})$.
Precompute: $\Delta_{1E} := \langle \mathbf{\Gamma}_{1E}, \mathbf{\Gamma}'_2 \rangle_e$, $\Delta_{1O} := \langle \mathbf{\Gamma}_{1O}, \mathbf{\Gamma}'_2 \rangle_e$, $\Delta_{2E} := \langle \mathbf{\Gamma}'_1, \mathbf{\Gamma}_{2E} \rangle_e$, $\Delta_{2O} := \langle \mathbf{\Gamma}'_1, \mathbf{\Gamma}_{2O} \rangle_e$ and $\chi := \langle \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \rangle_e$.

The prover computes and sends the necessary cross-products to the verifier:
1. $\mathcal{P}$ computes $D_{1E} := \langle \boldsymbol{X}_E, \mathbf{\Gamma}'_2 \rangle_e$, $D_{1O} := \langle \boldsymbol{X}_O, \mathbf{\Gamma}'_2 \rangle_e$, $D_{2E} := \langle \mathbf{\Gamma}'_1, \boldsymbol{Y}_E \rangle_e$ and $D_{2O} := \langle \mathbf{\Gamma}'_1, \boldsymbol{Y}_O \rangle_e$.
2. $\mathcal{P}$ also computes $E_{1\beta} := \langle \boldsymbol{\ell}, \mathbf{\Gamma}_1 \rangle_{\mathbb{G}}$, $E_{2\beta} := \langle \boldsymbol{r}, \mathbf{\Gamma}_2 \rangle_{\mathbb{G}}$.
3. $\mathcal{P}$ sends $D_{1E}, D_{1O}, D_{2E}, D_{2O}, E_{1\beta}, E_{2\beta}$ to $\mathcal{V}$.

4. $\mathcal{V}$ samples $\beta \xleftarrow{\$} \mathbb{F}$ and sends it to $\mathcal{P}$.

The prover folds the commitment key into its witness with respect to the challenge β:
5. $\mathcal{P}$ sets $\boldsymbol{X} = \boldsymbol{X} + \beta \mathbf{\Gamma}_1$ and $\boldsymbol{Y} = \boldsymbol{Y} + \beta^{-1} \mathbf{\Gamma}_2$.
6. $\mathcal{P}$ computes $C_+ := \langle \boldsymbol{X}_E, \boldsymbol{Y}_O \rangle_e$ and $C_- := \langle \boldsymbol{X}_O, \boldsymbol{Y}_E \rangle_e$.
7. $\mathcal{P}$ sets $E_{1+} := \langle \boldsymbol{\ell}_O, \boldsymbol{X}_E \rangle_{\mathbb{G}}$, $E_{1-} := \langle \boldsymbol{\ell}_E, \boldsymbol{X}_O \rangle_{\mathbb{G}}$, $E_{2+} := \langle \boldsymbol{r}_E, \boldsymbol{Y}_O \rangle_{\mathbb{G}}$, $E_{2-} := \langle \boldsymbol{r}_O, \boldsymbol{Y}_E \rangle_{\mathbb{G}}$.
8. $\mathcal{P}$ sends $C_+, C_-, E_{1+}, E_{1-}, E_{2+}, E_{2-}$ to $\mathcal{V}$.

9. $\mathcal{V}$ samples $\alpha \xleftarrow{\$} \mathbb{F}$ and sends it to $\mathcal{P}$.

The prover and verifier fold their instance and witness in half with respect to the challenge α:
10. $\mathcal{P}$ sets $\boldsymbol{X}' := \alpha \boldsymbol{X}_E + \boldsymbol{X}_O$ and $\boldsymbol{Y}' := \alpha^{-1} \boldsymbol{Y}_E + \boldsymbol{Y}_O$.
11. $\mathcal{P}$ and $\mathcal{V}$ set

$$
\begin{aligned}
C' &:= C + \chi + \beta D_2 + \beta^{-1} D_1 + \alpha C_+ + \alpha^{-1} C_-, \\
D'_1 &:= \alpha D_{1E} + D_{1O} + \alpha\beta \Delta_{1E} + \beta \Delta_{1O}, \\
D'_2 &:= \alpha^{-1} D_{2E} + D_{2O} + \alpha^{-1}\beta^{-1} \Delta_{2E} + \beta^{-1} \Delta_{2O}, \\
E'_1 &:= E_1 + \beta E_{1\beta} + \alpha E_{1+} + \alpha^{-1} E_{1-}, \\
E'_2 &:= E_2 + \beta^{-1} E_{2\beta} + \alpha E_{2+} + \alpha^{-1} E_{2-}, \\
\boldsymbol{\ell}' &:= \alpha^{-1} \boldsymbol{\ell}_E + \boldsymbol{\ell}_O, \\
\boldsymbol{r}' &:= \alpha \boldsymbol{r}_E + \boldsymbol{r}_O
\end{aligned}
$$

12. Define $\mathbb{i}' := (2^{n-1}, \mathbf{\Gamma}'_1, \mathbf{\Gamma}'_2)$, $\mathbb{x}' := (C', D'_1, D'_2, E'_1, E'_2, \boldsymbol{\ell}', \boldsymbol{r}')$ and $\mathbb{w}' := (\boldsymbol{X}', \boldsymbol{Y}')$.
13. $\mathcal{P}$ receives output $(\mathbb{i}', \mathbb{x}', \mathbb{w}')$ and $\mathcal{V}$ receives output $(\mathbb{i}', \mathbb{x}')$.

Lemma D.2 ([Lee21, Theorem 6]). *Let $\mathbf{\Gamma}'_1 \xleftarrow{\$} \mathbb{G}_1^{2^{n-1}}$, $\mathbf{\Gamma}'_2 \xleftarrow{\$} \mathbb{G}_2^{2^{n-1}}$ and $\mathbb{i} = (2^n, \mathbf{\Gamma}_1, \mathbf{\Gamma}_2)$ be an index for $\mathscr{R}_{\mathsf{SPP}}$. Then the following is an interactive argument of knowledge for proving that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathscr{R}_{\mathsf{SPP}}$:*
1. Prover and verifier run: $(\mathbb{i}', \mathbb{x}', \mathbb{w}') \leftarrow \mathsf{Dory.Reduce}(\langle \mathcal{P}((\mathbf{\Gamma}'_1, \mathbf{\Gamma}'_2), \mathbb{i}, \mathbb{x}, \mathbb{w}), \mathcal{V}((\mathbf{\Gamma}'_1, \mathbf{\Gamma}'_2), \mathbb{i}, \mathbb{x}) \rangle)$.
2. Prover sends $\mathbb{w}'$ to verifier.

[24] We note that in [Lee21], this protocol is actually called Dory.ReduceExt, but we will denote it Dory.Reduce for brevity.

3. Verifier accepts if and only if $(\mathbb{i}', \mathbb{x}', \mathbb{w}') \in \mathscr{R}_{\mathsf{SPP}}$.

D.2 Read-write streaming prover for Dory.Reduce

We now present a read-write streaming prover for Dory.Reduce.

Read-write Streaming Algorithm 16: prover for Dory.Reduce

$\mathsf{P}((\mathbf{R}_{\Gamma 1'}.\mathsf{init}(\boldsymbol{\Gamma}_1'), \mathbf{R}_{\Gamma 2'}.\mathsf{init}(\boldsymbol{\Gamma}_2')), \mathbb{i}, \mathbb{x}, \mathbb{w})$:
Parse: $\mathbb{i} = (2^n, \mathbf{R}_{\Gamma 1}.\mathsf{init}(\boldsymbol{\Gamma}_1), \mathbf{R}_{\Gamma 2}.\mathsf{init}(\boldsymbol{\Gamma}_2))$, $\mathbb{x} = (C, D_1, D_2, E_1, E_2, \mathbf{R}_\ell.\mathsf{init}(\boldsymbol{\ell}), \mathbf{R}_r.\mathsf{init}(\boldsymbol{r}))$ and $\mathbb{w} = (\mathbf{R}_X.\mathsf{init}(\boldsymbol{X}), \mathbf{R}_Y.\mathsf{init}(\boldsymbol{Y}))$.
Precompute: $\Delta_{1E} := \langle \boldsymbol{\Gamma}_{1E}, \boldsymbol{\Gamma}_2' \rangle_e$, $\Delta_{1O} := \langle \boldsymbol{\Gamma}_{1O}, \boldsymbol{\Gamma}_2' \rangle_e$, $\Delta_{2E} := \langle \boldsymbol{\Gamma}_1', \boldsymbol{\Gamma}_{2E} \rangle_e$, $\Delta_{2O} := \langle \boldsymbol{\Gamma}_1', \boldsymbol{\Gamma}_{2O} \rangle_e$ and $\chi := \langle \boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2 \rangle_e$.

Compute and send the necessary cross-products to the verifier:
1. Obtain $(\mathbf{R}_{XE}, \mathbf{R}_{XO}) \leftarrow \mathsf{SplitEO}(\mathbf{R}_X)$ and $(\mathbf{R}_{YE}, \mathbf{R}_{YO}) \leftarrow \mathsf{SplitEO}(\mathbf{R}_Y)$.
2. Obtain $(\mathbf{R}_{\ell E}, \mathbf{R}_{\ell O}) \leftarrow \mathsf{SplitEO}(2^n, \mathbf{R}_\ell)$ and $(\mathbf{R}_{rE}, \mathbf{R}_{rO}) \leftarrow \mathsf{SplitEO}(2^n, \mathbf{R}_r)$.
3. Compute $D_{1E} := \mathsf{InnerProd}(\mathbf{R}_{XE}, \mathbf{R}_{\Gamma 2'})$, $D_{1O} := \mathsf{InnerProd}(\mathbf{R}_{XO}, \mathbf{R}_{\Gamma 2'})$, $D_{2E} := \mathsf{InnerProd}(\mathbf{R}_{\Gamma 1'}, \mathbf{R}_{YE})$ and $D_{2O} := \mathsf{InnerProd}(\mathbf{R}_{\Gamma 1'}, \mathbf{R}_{YO})$.
4. Compute $E_{1\beta} := \mathsf{InnerProd}(\mathbf{R}_\ell, \mathbf{R}_{\Gamma 1})$, $E_{2\beta} := \mathsf{InnerProd}(\mathbf{R}_r, \mathbf{R}_{\Gamma 2})$.
5. Send $D_{1E}, D_{1O}, D_{2E}, D_{2O}, E_{1\beta}, E_{2\beta}$ to $\mathcal{V}$.

6. Receive $\beta \xleftarrow{\$} \mathbb{F}$ from $\mathcal{V}$.

Fold the commitment key into the witness with respect to the challenge β:
7. $\mathsf{LinComb}(1, \beta, \mathbf{R}_X, \mathbf{R}_{\Gamma 1})$ and $\mathsf{LinComb}(1, \beta^{-1}, \mathbf{R}_Y, \mathbf{R}_{\Gamma 2})$.
8. Compute $C_+ := \mathsf{InnerProd}(\mathbf{R}_{XE}, \mathbf{R}_{YO})$ and $C_- := \mathsf{InnerProd}(\mathbf{R}_{XO}, \mathbf{R}_{YE})$.
9. Compute $E_{1+} := \mathsf{InnerProd}(\mathbf{R}_{\ell O}, \mathbf{R}_{XE})$, $E_{1-} := \mathsf{InnerProd}(\mathbf{R}_{\ell E}, \mathbf{R}_{XO})$.
10. Compute $E_{2+} := \mathsf{InnerProd}(\mathbf{R}_{rE}, \mathbf{R}_{YO})$, $E_{2-} := \mathsf{InnerProd}(\mathbf{R}_{rO}, \mathbf{R}_{YE})$.
11. Send $C_+, C_-, E_{1+}, E_{1-}, E_{2+}, E_{2-}$ to $\mathcal{V}$.

12. Receive $\alpha \xleftarrow{\$} \mathbb{F}$ from $\mathcal{V}$.

Fold the instance and witness in half with respect to the challenge α:
13. $\mathsf{kFoldInPlace}(2^n, \alpha, 1, 1, \mathbf{R}_X)$ and $\mathsf{kFoldInPlace}(2^n, \alpha^{-1}, 1, 1, \mathbf{R}_Y)$.
14. $\mathsf{kFoldInPlace}(2^n, \alpha^{-1}, 1, 1, \mathbf{R}_\ell)$ and $\mathsf{kFoldInPlace}(2^n, \alpha, 1, 1, \mathbf{R}_r)$.
15. Set

$$\begin{aligned} C' &:= C + \chi + \beta D_2 + \beta^{-1} D_1 + \alpha C_+ + \alpha^{-1} C_- \\ D_1' &:= \alpha D_{1E} + D_{1O} + \alpha\beta \Delta_{1E} + \beta \Delta_{1O} \\ D_2' &:= \alpha^{-1} D_{2E} + D_{2O} + \alpha^{-1}\beta^{-1} \Delta_{2E} + \beta^{-1} \Delta_{2O}, \\ E_1' &:= E_1 + \beta E_{1\beta} + \alpha E_{1+} + \alpha^{-1} E_{1-}, \\ E_2' &:= E_2 + \beta^{-1} E_{2\beta} + \alpha E_{2+} + \alpha^{-1} E_{2-}, \end{aligned}$$

16. Define $\mathbb{i}' := (2^{n-1}, \mathbf{R}_{\Gamma 1'}, \mathbf{R}_{\Gamma 2'})$, $\mathbb{x}' := (C', D_1', D_2', E_1', E_2', \mathbf{R}_\ell, \mathbf{R}_r)$ and $\mathbb{w}' := (\mathbf{R}_X, \mathbf{R}_Y)$.
17. Retain output $(\mathbb{i}', \mathbb{x}', \mathbb{w}')$.

Lemma D.3. Dory.Reduce *and Algorithm 16 together define a read-write streaming reduction of knowledge with the following properties:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(\sqrt{N})$	$O(N)$	$O(\log N)$	$O(N)$	$O(\log N)$	$O(\log N)$

The proof of this lemma is similar to that of Lemma 6.1.

D.3 Dory-InnerProduct

Just like the GIPA protocol from Appendix B.2.3, the full argument for $\mathscr{R}_{\mathsf{SPP}}$, Dory.InnerProd, applies Dory.Reduce iteratively to shrink the size of the scalar pairing product instance to length 1, and then checks the instance directly.

Lemma D.4. Dory.InnerProd *is a read-write streaming interactive argument of knowledge for* $\mathscr{R}_{\mathsf{SPP}}$ *with the following properties:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(N)$	$O(N)$	$O(\log N)$	$O(N)$	$O(N)$	$O(\log N)$

Proof. Completeness and knowledge soundness follow from [Lee21, Theorem 7]. The existence and efficiency of a read-write streaming prover for Dory.InnerProd follows from Lemma D.3. □

D.4 VMV from Dory-Innerproduct

In this section we describe how to build a VMV protocol from Dory.InnerProd. Notice that VMV.Setup precomputes all the necessary terms for Dory.InnerProd, and that VMV.Open lifts $\boldsymbol{a} := \boldsymbol{\ell}^T\mathbf{M}$ into a vector in $\mathbb{G}_2$ using $\Gamma_{2,\mathsf{fin}}$ (this is highlighted in blue).

$\mathsf{VMV.Setup}(1^\lambda, N^2 = 2^{2n}) \to (\mathsf{ck}, \mathsf{vk})$:

1. Sample $\mathbf{\Gamma}_{1,1} \xleftarrow{\$} \mathbb{G}_1^N$ and $(\mathbf{\Gamma}_{2,1}, \Gamma_{2,\mathsf{fin}}) \xleftarrow{\$} \mathbb{G}_2^N \times \mathbb{G}_2$.
2. For i in $[1, \ldots, n]$ set:
3. $\quad \mathbf{\Gamma}_{1,i+1} := (\mathbf{\Gamma}_{1,i})_L$ and $\mathbf{\Gamma}_{2,i+1} := (\mathbf{\Gamma}_{2,i})_L$.
4. $\quad \chi_i := \langle \mathbf{\Gamma}_{1,i}, \mathbf{\Gamma}_{2,i} \rangle_e$.
5. $\quad \Delta_{1E,i} := \langle (\mathbf{\Gamma}_{1,i})_E, \mathbf{\Gamma}_{2,i+1} \rangle_e$, $\Delta_{1O,i} := \langle (\mathbf{\Gamma}_{1,i})_O, \mathbf{\Gamma}_{2,i+1} \rangle_e$, $\Delta_{2E,i} := \langle \mathbf{\Gamma}_{1,i+1}, (\mathbf{\Gamma}_{2,i})_E \rangle_e$ and $\Delta_{2O,i} := \langle \mathbf{\Gamma}_{1,i+1}, (\mathbf{\Gamma}_{2,i})_O \rangle_e$.
6. Compute $\chi_{n+1} := e(\Gamma_{1,n+1}, \Gamma_{2,n+1})$.
7. Set $\mathsf{precomp} := ([\chi_i]_{i=1}^{n+1}, [\Delta_{1E,i}]_{i=1}^n, [\Delta_{1O,i}]_{i=1}^n, [\Delta_{2E,i}]_{i=1}^n, [\Delta_{2O,i}]_{i=1}^n, \Gamma_{2,\mathsf{fin}})$.
8. Set $\mathsf{ck} := ([\mathbf{\Gamma}_{1,i}]_{i=1}^{n+1}, [\mathbf{\Gamma}_{2,i}]_{i=1}^{n+1}, \mathsf{precomp})$.
9. Set $\mathsf{vk} := (\Gamma_{1,n+1}, \Gamma_{2,n+1}, \mathsf{precomp})$.
10. Output $(\mathsf{ck}, \mathsf{vk})$.

$\mathsf{VMV.Commit}(\mathsf{ck}, \mathbf{M}) \to C_{\mathbf{M}}$:

1. Parse ck as $([\mathbf{\Gamma}_{1,i}]_{i=1}^{n+1}, [\mathbf{\Gamma}_{2,i}]_{i=1}^{n+1}, \mathsf{precomp})$.
2. For all $i \in [0, 1, \ldots, N-1]$: Commit to the i-th row as $C_i \leftarrow \mathsf{CM}_P.\mathsf{Commit}(\mathbf{\Gamma}_{1,1}, \mathbf{M}_i)$.
3. Output $C_{\mathbf{M}} \leftarrow \mathsf{CM}_1.\mathsf{Commit}(\mathbf{\Gamma}_{2,1}, \boldsymbol{C})$.

$\mathsf{VMV.Open}(\langle \mathcal{P}(\mathsf{ck}, \mathbb{x}, \mathbf{M}), \mathcal{V}(\mathsf{vk}, \mathbb{x})\rangle)$:
Parse: $\mathsf{ck} = ([\mathbf{\Gamma}_{1,i}]_{i=1}^{n+1}, [\mathbf{\Gamma}_{2,i}]_{i=1}^{n+1}, \mathsf{precomp})$, $\mathsf{vk} = (\Gamma_{1,n+1}, \Gamma_{2,n+1}, \mathsf{precomp})$ and $\mathbb{x} = (C_{\mathbf{M}}, \boldsymbol{\ell}, \boldsymbol{r}, y)$.
1. For all $i \in [0, 1, \ldots, N-1]$: $\mathcal{P}$ computes $C_i := \mathsf{CM}_P.\mathsf{Commit}(\mathbf{\Gamma}_{1,1}, \mathbf{M}_i)$.
2. $\mathcal{P}$ sets $\boldsymbol{X} := \boldsymbol{C}$, $D_1 := C_{\mathbf{M}}$.
3. $\mathcal{P}$ computes $\boldsymbol{a} := \boldsymbol{\ell}^T \mathbf{M}$, and $\boldsymbol{Y} := \boldsymbol{a} \cdot \Gamma_{2,\mathsf{fin}}$.
4. $\mathcal{P}$ sets
$$C := \langle \boldsymbol{X}, \boldsymbol{Y}\rangle_e,$$
$$D_1 := C_{\mathbf{M}}, \qquad D_2 := \mathsf{CM}_2(\mathbf{\Gamma}_{1,1}, \boldsymbol{Y}),$$
$$E_1 := \langle \boldsymbol{\ell}, \boldsymbol{X}\rangle_{\mathbb{G}}, \quad E_2 := \langle \boldsymbol{r}, \boldsymbol{Y}\rangle_{\mathbb{G}}.$$
5. $\mathcal{P}$ sends C, D_2, E_1, E_2 to $\mathcal{V}$.
6. $\mathcal{V}$ checks that $E_2 = y \cdot \Gamma_{2,\mathsf{fin}}$.
7. $\mathcal{V}$ checks that $e(E_1, \Gamma_{2,\mathsf{fin}}) = D_2$.
8. Set $\mathbb{i}_{\mathsf{SPP}} := (N, \mathbf{\Gamma}_{1,1}, \mathbf{\Gamma}_{2,1})$, $\mathbb{x}_{\mathsf{SPP}} := (C, D_1, D_2, E_1, E_2, \boldsymbol{\ell}, \boldsymbol{r})$ and $\mathbb{w}_{\mathsf{SPP}} := (\boldsymbol{X}, \boldsymbol{Y})$.
9. $\mathcal{P}$ and $\mathcal{V}$ run $\mathsf{Dory.InnerProd}(\langle \mathcal{P}(\mathbb{i}_{\mathsf{SPP}}, \mathbb{x}_{\mathsf{SPP}}, \mathbb{w}_{\mathsf{SPP}}), \mathcal{V}(\mathbb{i}_{\mathsf{SPP}}, \mathbb{x}_{\mathsf{SPP}})\rangle)$.

Lemma D.5. VMV *is a read-write streaming VMV argument for 'random evaluation vectors'*[25] *with the following efficiency:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(\sqrt{N})$	$O(N)$	$O(\log N)$	$O(N)$	$O(\log N)$	$O(\log N)$

Proof. The completeness and soundness claims follows from [Lee21, Theorem 9], while the efficiency claims follow from the efficiency of our read-write streaming algorithm for Dory.InnerProd. □

Using VMV to construct a PC scheme as described in Appendix C.2 yields the following lemma.

Lemma D.6. *The* Dory *PC scheme [Lee21] is a read-write streaming polynomial commitment scheme for n-variate multilinear polynomials with the following efficiency:*

SRS size	prover time	prover space		check time	proof size
		random-access	streaming		
$O(\sqrt{N})$	$O(N)$	$O(\log N)$	$O(N)$	$O(\log N)$	$O(\log N)$

[25] The current scheme is only sound for $\boldsymbol{\ell} := \otimes_{i=1}^{n/2}(1 - z_i, z_i)$ and $\boldsymbol{r} := \otimes_{i=n/2+1}^{n}(1 - z_i, z_i)$ constructed using a random $\boldsymbol{z} \xleftarrow{\$} \mathbb{F}^n$. [Lee21] shows how to remedy this by repeating the VMV argument twice, once with random evaluation vectors and once with the required evaluation vectors.

References

[AFGHO16] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo. "Structure-Preserving Signatures and Commitments to Group Elements". In: *Journal of Cryptology* 29.2 (2016), pp. 363–421.

[AST24] A. Arun, S. T. V. Setty, and J. Thaler. "Jolt: SNARKs for Virtual Machines via Lookups". In: *Proceedings of the 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '24. 2024, pp. 3–33.

[AV88] A. Aggarwal and J. S. Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127.

[BBBPWM18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *Proceedings of the 39th IEEE Symposium on Security and Privacy*. S&P '18. 2018, pp. 315–334.

[BBHV22] L. Bangalore, R. Bhadauria, C. Hazay, and M. Venkitasubramaniam. "On Black-Box Constructions of Time and Space Efficient Sublinear Arguments from Symmetric-Key Primitives". In: *Proceedings of the 20th Theory of Cryptography Conference*. TCC '22. 2022, pp. 417–446.

[BC12] N. Bitansky and A. Chiesa. "Succinct Arguments from Multi-Prover Interactive Proofs and their Efficiency Benefits". In: *Proceedings of the 32nd Annual International Cryptology Conference*. CRYPTO '12. 2012, pp. 255–272.

[BCCGP16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. "Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting". In: *Proceedings of the 35th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT '16. 2016, pp. 327–357.

[BCCT12] N. Bitansky, R. Canetti, A. Chiesa, and E. 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 '12. 2012, pp. 326–349.

[BCCT13] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. "Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data". In: *Proceedings of the 45th ACM Symposium on the Theory of Computing*. STOC '13. 2013, pp. 111–120.

[BCGJM18] J. Bootle, A. Cerulli, J. Groth, S. K. Jakobsen, and M. Maller. "Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution". In: *Proceedings of the 24th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT '18. 2018, pp. 595–626.

[BCHO22] J. Bootle, A. Chiesa, Y. Hu, and M. Orrù. "Gemini: Elastic SNARKs for Diverse Environments". In: *Proceedings of the 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '22. 2022.

[BCLMS21] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner. "Proof-Carrying Data Without Succinct Arguments". In: *Proceedings of the 41st Annual International Cryptology Conference*. CRYPTO '21. 2021, pp. 681–710.

[BCMS20] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. "Proof-Carrying Data from Accumulation Schemes". In: *Proceedings of the 18th Theory of Cryptography Conference*. TCC '20. 2020.

[BCS21] J. Bootle, A. Chiesa, and K. Sotiraki. "Sumcheck Arguments and Their Applications". In: *Proceedings of the 41st Annual International Cryptology Conference*. CRYPTO '21. 2021, pp. 742–773.

[BCTV14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves". In: *Proceedings of the 34th Annual International Cryptology Conference*. CRYPTO '14. 2014, pp. 276–294.

[BDFG21] D. Boneh, J. Drake, B. Fisch, and A. Gabizon. "Halo Infinite: Proof-Carrying Data from Additive Polynomial Commitments". In: *Proceedings of the 41st Annual International Cryptology Conference*. CRYPTO '21. 2021, pp. 649–680.

[BFS20] B. Bünz, B. Fisch, and A. Szepieniec. "Transparent SNARKs from DARK Compilers". In: *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '20. 2020, pp. 677–706.

[BGH19] S. Bowe, J. Grigg, and D. Hopwood. "Halo: Recursive Proof Composition without a Trusted Setup". IACR ePrint Report 2019/1021. 2019.

[BHRRS20] A. R. Block, J. Holmgren, A. Rosen, R. D. Rothblum, and P. Soni. "Public-Coin Zero-Knowledge Arguments with (almost) Minimal Time and Space Overheads". In: *Proceedings of the 18th Theory of Cryptography Conference*. TCC '20. 2020.

[BHRRS21] A. R. Block, J. Holmgren, A. Rosen, R. D. Rothblum, and P. Soni. "Time- and Space-Efficient Arguments from Groups of Unknown Order". In: *Proceedings of the 41st Annual International Cryptology Conference*. CRYPTO '21. 2021, pp. 123–152.

[BJR07] P. Beame, T. S. Jayram, and A. Rudra. "Lower Bounds for Randomized Read/Write Stream Algorithms". In: *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*. STOC '07. 2007.

[BMMTV21] B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. "Proofs for Inner Pairing Products and Applications". In: *Proceedings of the 27th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT '21. 2021, pp. 65–97.

[CBBZ23] B. Chen, B. Bünz, D. Boneh, and Z. 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 '23. 2023, pp. 499–530.

[CFFZ24] A. Chiesa, E. Fedele, G. Fenzi, and A. Zitek-Estrada. "A Time-Space Tradeoff for the Sumcheck Prover". IACR ePrint Report 2024/524. 2024.

[CHMMVW20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. 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 '20. 2020.

[CM24] J. Cook and D. Moshkovitz. "Explicit Time and Space Efficient Encoders Exist Only with Random Access". In: *Proceedings of the 39th Computational Complexity Conference*. CCC '24. 2024, 5:1–5:54.

[con22] arkworks contributors. `arkworks` *zkSNARK ecosystem*. 2022.

[CT10] A. Chiesa and E. Tromer. "Proof-Carrying Data and Hearsay Arguments from Signature Cards". In: *Proceedings of the 1st Symposium on Innovations in Computer Science*. ICS '10. 2010, pp. 310–331.

[DFR09] C. Demetrescu, I. Finocchi, and A. Ribichini. "Trading off space for passes in graph streaming problems". In: *ACM Trans. Algorithms* 6.1 (2009), 6:1–6:17.

[FJM14] N. François, R. Jain, and F. Magniez. "Unidirectional Input/Output Streaming Complexity of Reversal and Sorting". In: *Proceedings of the 17th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and the 18th International Workshop on Randomization and Computation*. APPROX/RANDOM '14. 2014, pp. 654–668.

[FS86] A. Fiat and A. Shamir. "How to prove yourself: practical solutions to identification and signature problems". In: *Proceedings of the 6th Annual International Cryptology Conference*. CRYPTO '86. 1986, pp. 186–194.

[GHS06] M. Grohe, A. Hernich, and N. Schweikardt. "Randomized computations on large data sets: tight lower bounds". In: *Proceedings of the 25th Symposium on Principles of Database Systems*. PODS '06. 2006, pp. 243–252.

[GKS05] M. Grohe, C. Koch, and N. Schweikardt. "Tight Lower Bounds for Query Processing on Streaming and External Memory Data". In: *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*. ICALP '05. 2005, pp. 1076–1088.

[GLSTW23] A. Golovnev, J. Lee, S. T. V. Setty, J. Thaler, and R. S. Wahby. "Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS". In: *Proceedings of the 43rd Annual International Cryptology Conference*. CRYPTO '23. 2023, pp. 193–226.

[GS05] M. Grohe and N. Schweikardt. "Lower Bounds for Sorting with Few Random Accesses to External Memory". In: *Proceedings of the 24th ACM Symposium on Principles of Database Systems*. PODS '05. 2005.

[GW19] A. Gabizon and Z. J. Williamson. "The turbo-plonk program syntax for specifying snark programs". Preprint. 2019.

[Hab22] U. Haböck. "Multivariate lookups based on logarithmic derivatives". IACR ePrint Report 2022/1530. 2022.

[HLP24] U. Haböck, D. Levit, and S. Papini. "Circle STARKs". IACR ePrint Report 2024/278. 2024.

[KP23] A. Kothapalli and B. Parno. "Algebraic Reductions of Knowledge". In: *Proceedings of the 43rd Annual International Cryptology Conference*. CRYPTO '23. 2023, pp. 669–701.

[KST22] A. Kothapalli, S. T. V. Setty, and I. Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: *Proceedings of the 42nd Annual International Cryptology Conference*. CRYPTO '22. 2022, pp. 359–388.

[KZG10] A. Kate, G. M. Zaverucha, and I. 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 '10. 2010, pp. 177–194.

[Lee21] J. Lee. "Dory: Efficient, Transparent Arguments for Generalised Inner Products and Polynomial Commitments". In: *Proceedings of the 19th Theory of Cryptography Conference*. TCC '21. 2021, pp. 1–34.

[LFKN92] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. "Algebraic Methods for Interactive Proof Systems". In: *Journal of the ACM* 39.4 (1992), pp. 859–868.

[LMR19] R. W. F. Lai, G. Malavolta, and V. Ronge. "Succinct Arguments for Bilinear Group Arithmetic: Practical Structure-Preserving Cryptography". In: *Proceedings of the 26th ACM Conference on Computer and Communications Security*. CCS '19. 2019, pp. 2057–2074.

[NDCTB24] W. Nguyen, T. Datta, B. Chen, N. Tyagi, and D. Boneh. "Mangrove: A Scalable Framework for Folding-based SNARKs". In: *Proceedings of the 44th Annual International Cryptology Conference*. CRYPTO '24. 2024.

[Pip80] N. Pippenger. "On the Evaluation of Powers and Monomials". In: *SIAM Journal on Computing* 9.2 (1980), pp. 230–250.

[Pol] Polygon. *Plonky3*. URL: `https://github.com/plonky3/plonky3`.

[PP24] C. Pappas and D. Papadopoulos. "Sparrow: Space-Efficient zkSNARK for Data-Parallel Circuits and Applications to Zero-Knowledge Decision Trees". IACR ePrint Report 2024/1631. 2024.

[PST13] C. Papamanthou, E. Shi, and R. Tamassia. "Signatures of Correct Computation". In: *Proceedings of the 10th Theory of Cryptography Conference*. TCC '13. 2013, pp. 222–242.

[PY14] P. A. Papakonstantinou and G. Yang. "Cryptography with Streaming Algorithms". In: *Proceedings of the 34th Annual Cryptology Conference*. CRYPTO '14. 2014, pp. 55–70.

[Rot24] R. D. Rothblum. "A Note on Efficient Computation of the Multilinear Extension". IACR ePrint Report 2024/1103. 2024.

[Set20] S. Setty. "Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup". In: *Proceedings of the 40th Annual International Cryptology Conference*. CRYPTO '20. 2020, pp. 704–737.

[SL20] S. T. V. Setty and J. Lee. "Quarks: Quadruple-efficient transparent zkSNARKs". IACR ePrint Report 2020/1275. 2020.

[Tha13] J. Thaler. "Time-Optimal Interactive Proofs for Circuit Evaluation". In: *Proceedings of the 33rd Annual International Cryptology Conference*. CRYPTO '13. 2013, pp. 71–89.

[Tha22] J. Thaler. "Proofs, Arguments, and Zero-Knowledge". In: *Found. Trends Priv. Secur.* 4.2-4 (2022), pp. 117–660.

[Whi18] B. WhiteHat. "roll_up: A Scalable Zero Knowledge Roll Up". Accessed: 2024-02-10. 2018.

[WHV24] R. Wang, C. Hazay, and M. Venkitasubramaniam. "Ligetron: Lightweight Scalable End-to-End Zero-Knowledge Proofs. Post-Quantum ZK-SNARKs on a Browser". In: *Proceedings of the 45th IEEE Symposium on Security and Privacy*. IEEE S&P '24. 2024.

[WTSTW18] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. "Doubly-Efficient zkSNARKs Without Trusted Setup". In: *Proceedings of the 39th IEEE Symposium on Security and Privacy*. S&P '18. 2018, pp. 926–943.

[Xie+22] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song. "zkBridge: Trustless Cross-chain Bridges Made Practical". In: *Proceedings of the 29th ACM Conference on Computer and Communications Security*. CCS '22. 2022, pp. 3003–3017.

[XZZPS19] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation". In: *Proceedings of the 39th Annual International Cryptology Conference*. CRYPTO '19. 2019, pp. 733–764.

[ZCLKZ24] S. Zhang, D. Cai, Y. Li, H. Kan, and L. Zhang. "Epistle: Elastic Succinct Arguments for Plonk Constraint System". IACR ePrint Report 2024/872. 2024.

[ZGKPP18] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vRAM: Faster Verifiable RAM with Program-Independent Preprocessing". In: *Proceedings of the 39th IEEE Symposium on Security and Privacy*. IEEE S&P '18. 2018, pp. 908–925.