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

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. 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 less than 750 MB of memory and incurring only 10% proving latency overhead compared to a state-of-the-art memory-intensive baseline (HyperPlonk [EUROCRYPT 2023]) that requires much more memory.

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 [53] and cryptocurrency bridges [54]. 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 [15, 47, 24, 31, 33, 46, 3] 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 smaller) space complexity of the computation. For example, the HyperPlonk SNARK [24] requires over 16 GB of RAM to prove that 10 kB of data was hashed correctly with SHA256.

This has motivated recent efforts to reduce these memory requirements. These efforts proceed by reducing the size of the prover's internal state via disparate techniques such as complexity-preserving SNARKs [10, 9, 4], streaming SNARKs [11, 12, 16, 56], and recursive composition [7, 20, 13, 35]. However, as we detail in Section 8, these approaches achieve lower memory usage only by sacrificing asymptotic and concrete prover time. Moreover, some approaches even seem to require an inherent space-time tradeoff [4, 27].

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 is 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.

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

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, this incurs fresh challenges of its own: for efficiency, disk operations read and write data in large blocks, and each operation is much more costly than a RAM operation. Hence, a naive attempt to port an algorithm to our model could incur significant latency overheads.

To avoid this, our model restricts the algorithm's disk accesses to follow a predictable and data-independent *streaming* access pattern. This 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 [24], 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 → SNARK" paradigm [25, 22] 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 [42, 18, 51, 23, 37]. All our PIOP and PC constructions preserve the time complexity of their non-streaming counterparts, while reducing their random-access space to logarithmic. We additionally provide algorithmic optimizations that minimize the amount of I/O required for key building blocks of our PIOPs and PC schemes.

Implementation. We implement Scribe as a Rust library based on the `arkworks` framework [26]. 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 optimizations such as prefetching, batch operations, and parallelization. We provide details about our implementation and optimizations in Section 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 h on a server-class machine. Our evaluation demonstrates that assuming reasonable hardware, read-write streaming imposes minimal overheads (10%) over the memory-intensive baseline of HyperPlonk. Furthermore, compared to a prior state-of-the-art low-memory SNARK that operates in the same application regime [16], 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 of size 2^{24} gates, 32× 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.

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 Notation and preliminaries

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 $[x_k]_{k=i}^j$. We write $x \xleftarrow{\$} S$ to denote 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 MSB binary decomposition of i, 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 denote by $\mathsf{int}(\boldsymbol{x})$ the inverse operation which maps $\boldsymbol{x} = \mathsf{bin}_n(v)$ to v in the obvious way.

Multilinear polynomials. A polynomial is *multilinear* if the degree of each variable in every monomial is at most

[3] We make an exception when indexing vectors corresponding to coefficients of multilinear polynomials; these are zero-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.

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

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 $p(\boldsymbol{x}) \in \mathbb{F}$.

Algebraic notation. We use additive notation for groups. We use bilinear groups sampled by an algorithm $\mathsf{SampleGrp}$ that outputs $(\mathbb{F},\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e,G,H)$ where $\mathbb{F}$ is a prime field of size 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 $\langle \boldsymbol{x},\boldsymbol{y}\rangle_{\mathbb{F}} := \sum_{i=1}^N x_iy_i$, (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.

2.1 Background: HyperPlonk

To construct an efficient read-write streaming SNARK, our starting point will be the memory-*intensive* SNARK of HyperPlonk [24]. 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 a circuit of size N.
- concretely, HyperPlonk achieves better prover times than almost all prior SNARKs, and additionally supports attractive features such as custom gates [30].

SNARKs from Polynomial IOPs and PC schemes. HyperPlonk and Scribe are obtained via a popular methodology [25, 22] 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 [25]. 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, and v is the claimed value of $p(z)$.

The 'PIOP + PC' methodology 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 [29] on this interactive argument.

HyperPlonk [24] 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 add gate, then $o_i = \ell_i + r_i$; if it is a multiply gate, then $o_i = \ell_i \cdot r_i$.
2. *Wiring constraints*: enforce that wires between gates are connected correctly. E.g., if the output of gate j is the left input of gate i, then a wiring constraint enforces $\ell_i = o_j$.

To encode this circuit representation in polynomial form, HyperPlonk associates with each circuit the following vectors: (a) a *selector* $\boldsymbol{s} \in \mathbb{F}^m$ such that s_i is 0 if the i-th gate is an add gate, and is 1 if it is a multiply 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 if instead the right input is the output of gate j then $\sigma_i = j$.[7]

[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 [31] because some group operations, in particular multi-scalar multiplications, are asymptotically super-linear even with the best known algorithms [45].

[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 the full version of this paper 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

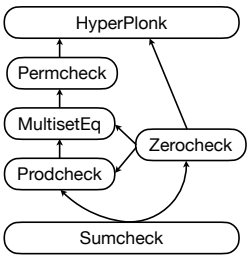

Figure 1: Components of the HyperPlonk PIOP.

HyperPlonk's PIOP. In a preprocessing phase, C is arithmetized by encoding $\boldsymbol{s}$, $\boldsymbol{\pi}$, $\boldsymbol{\sigma}$ 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 $\hat{\pi},\hat{\sigma}$ 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 5 and 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 [42]. We provide details about this scheme in Section 7.2, 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.

with both left and right inputs. We provide a formal definition of the relation which lifts these restrictions in the full version of the paper.

[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 the full version of this paper describes the indexer's task in detail.

3 Read-write streaming algorithms

We define read-write (RW) streams [32, 6, 41] in detail in Definition 3.1, and describe the behavior of RW streaming algorithms in Definition 3.2.

Definition 3.1. *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.* $m = \mathsf{r}$*) or write mode (i.e.* $m = \mathsf{w}$*), and* ℓ *is the length of the underlying ordered set. Read-write streams support the following operations:*

- $\mathbf{S}.\mathsf{read}()$*: if* $\mathbf{S}.m = \mathsf{r}$*, return the element of* $\mathbf{S}.\boldsymbol{a}$ *at the current position of* $\mathbf{S}.p$*, and move* $\mathbf{S}.p$ *to the next element of* $\mathbf{S}.\boldsymbol{a}$*.*
- $\mathbf{S}.\mathsf{write}(w)$*: if* $\mathbf{S}.m = \mathsf{w}$*, write* w *at the current position of* $\mathbf{S}.p$*, and move* $\mathbf{S}.p$ *to the next element of* $\mathbf{S}.\boldsymbol{a}$*.*
- $\mathbf{S}.\mathsf{init}(N)$*: initialize* $\mathbf{S} = (\boldsymbol{a}, p, \mathsf{w}, N)$ *in write mode by allocating a contiguous space for an array of* N *elements* $\boldsymbol{a}$*, and initialize* $\mathbf{S}.p$ *at the beginning of the allocated space. This space is not necessarily initialized with 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* $\mathbf{S}.\ell$*.*

Definition 3.2. *A read-write (RW) streaming algorithm* $\mathcal{A}(\mathbf{I}) \mapsto \mathbf{O}$ *is an algorithm that*

- *takes as input a read-only stream* $\mathbf{I}$ *of length* N *containing its input, and a (mutable reference to an) RW stream* $\mathbf{O}$ *onto which it writes its output.*
- *can use a sublinear amount* $m(N)$ *of random-access space.*
- *can allocate intermediate streams using* init *and perform read/write operations on them in a streaming manner using* $\mathsf{read}()$*,* $\mathsf{restart}()$*, and* $\mathsf{write}(\cdot)$*.*

Henceforth, when we say streaming*, we mean read-write streaming unless otherwise specified.*

The *random-access space complexity* of $\mathcal{A}$ is the amount of random-access space $m(N)$ allocated by $\mathcal{A}$. The *streaming space complexity* of $\mathcal{A}$ is $\sum_{i=1}^{k} \ell_i$ 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.

Restrictions of our model. Definition 3.2 places some restrictions on RW streaming algorithms: namely, it assumes that input streams are immutable, and that algorithms have a single input and output stream. These restrictions are without loss of generality, as detailed in Appendix F

The external memory model. The external memory (EM) model [2], like the read-write streaming model, captures algorithms whose inputs (and intermediate state) are too large to fit into a computer's main memory at once, and are hence stored

in external memory. Unlike the RW streaming model, the EM model allows algorithms *random-access* to the external memory. However, these accesses must occurs in large batches, and the I/O cost of the algorithm is measured in terms of the number of such batches. By batching I/O, an EM algorithm amortizes I/O cost over many elements.

We show in Appendix E that, even though read-write streaming algorithms perform I/O one element at a time, their streaming nature enables straightforward batching, and hence also good EM I/O complexity.

3.1 Composition of RW streaming algorithms

Let $\mathcal{A}(\mathbf{I}_1) \mapsto \mathbf{O}_1$ and $\mathcal{B}(\mathbf{I}_2) \mapsto \mathbf{O}_2$ be two RW streaming algorithms. We say that $\mathcal{C}(\mathbf{I}_1) \mapsto \mathbf{O}_2$ is the *composition* of $\mathcal{A}$ and $\mathcal{B}$ if it invokes $\mathcal{A}$ to obtain the input for $\mathcal{B}$, and then invokes $\mathcal{B}$ on the latter to obtain the final output. That is, it: (i) allocates space for $\mathbf{O}_1$; (ii) calls $\mathcal{A}(\mathbf{I}_1) \mapsto \mathbf{O}_1$; (iii) once $\mathcal{A}$ terminates, calls $\mathbf{O}_1.\mathsf{swapmode}()$; and (iv) invokes $\mathcal{B}(\mathbf{O}_1) \mapsto \mathbf{O}_2$.

This notion of composition can be used to implement RW streaming algorithms that call other RW streaming algorithms as subroutines. We say that $\mathcal{A}$ calls $\mathcal{B}$ as a subroutine if $\mathcal{A}$ can be split into two parts, $\mathcal{A}_1$ and $\mathcal{A}_2$, so that $\mathcal{A}$ is the composition of $\mathcal{A}_1, \mathcal{B}$, and $\mathcal{A}_2$. This naturally extends to multiple subroutine calls. The cost of subroutine calls is detailed below:

Lemma 3.3 (complexity of subroutine calls)**.** *Let $\mathcal{A}, \mathcal{B}$ be read-write streaming algorithms. For each algorithm $\mathcal{X} \in \{\mathcal{A}, \mathcal{B}\}$ let its time complexity be $t_{\mathcal{X}}$, its random-access space complexity be $m_{\mathcal{X}}$, and its streaming space complexity be $m_{\mathcal{X}}$. Let L be the number of times that $\mathcal{A}$ calls $\mathcal{B}$ as a subroutine. 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 3.4 (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 $\mathcal{C}(\mathbf{S}) \mapsto \mathbf{O}$. As stated, our model requires that $\mathcal{A}$ allocate an intermediate stream $\mathbf{S}$ that is provided as input to both $\mathcal{B}$ and $\mathcal{C}$. However, if $\mathcal{C}$ only makes a single pass over $\mathbf{S}$, then we can implement $\mathcal{A}$ without allocating space for $\mathbf{S}$ at all by directly "piping" the output of $\mathcal{B}$ to $\mathcal{C}$, similar to the composition of read-only streaming algorithms. We denote that such an optimization is taking place via the arrow '$\rightsquigarrow$'. With this notation, $\mathcal{A}$ can be written as $\mathcal{A} := \mathbf{O} \leftarrow \mathcal{C} \rightsquigarrow \mathcal{B} \rightsquigarrow \mathbf{I}$.

3.2 Common RW streaming subroutines

We define below some helper functions that we use in our 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 $\mathbf{W}$ whose i-th element is a tuple of k elements, where the j-th element of the tuple is the i-th element of $\mathbf{R}_j$.
- 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 accumulator s, and a function f, and produces a single element that is the result of reducing the elements of $\mathbf{R}$ using f.
- $\mathsf{SplitLSB}$ takes as input a stream $\mathbf{R}$ of length $N = 2^n$ and a parameter k, and splits $\mathbf{R}$ into 2^k output streams, each with $N/2^k$ elements, such that the i-th output stream contains the elements of $\mathbf{R}$ whose indices have the lowest k bits equal to i. $\mathsf{SplitEO}$ is the special case when $k = 1$ that splits $\mathbf{R}$ into two streams containing even and odd-numbered elements.

$\mathsf{Zip}(\mathbf{R}_1, \ldots, \mathbf{R}_k) \mapsto \mathbf{W}$:
1. For i in $[1, \ldots, \mathbf{R}_1.\mathsf{len}()]$, $\mathbf{W}.\mathsf{write}((\mathbf{R}_1.\mathsf{read}(), \ldots, \mathbf{R}_k.\mathsf{read}()))$.

$\mathsf{Map}(\mathbf{R}, f) \mapsto \mathbf{W}$:
1. For i in $[1, \ldots, \mathbf{R}.\mathsf{len}()]$:
2. $\quad \mathbf{W}.\mathsf{write}(f(\mathbf{R}.\mathsf{read}()))$.

$\mathsf{Reduce}(\mathbf{R}, f, s) \mapsto s$:
1. For i in $[1, \ldots, \mathbf{R}.\mathsf{len}()]$:
2. $\quad s \leftarrow f(s, \mathbf{R}.\mathsf{read}())$.

$\mathsf{SplitLSB}(\mathbf{R}, k) \mapsto (\mathbf{W}_1, \ldots, \mathbf{W}_{2^k})$:
1. For i in $[1, \ldots, \mathbf{R}.\mathsf{len}()/2^k]$: For j in $[1, \ldots, 2^k]$: $\mathbf{W}_j.\mathsf{write}(\mathbf{R}.\mathsf{read}())$.

4 Constructing RW streaming SNARKs

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 2.1): $\mathcal{P}$ 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 3.3 that supports multiple subroutines gives the following lemma:

Lemma 4.1. *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* $= t_{\mathsf{PIOP}}(N) + \mathsf{c}\cdot t_{\mathsf{C}}(N) + \mathsf{o}\cdot t_{\mathsf{O}}(N)$.
- *Random-access space* $= m_{\mathsf{PIOP}}(N) + m_{\mathsf{C}}(N) + m_{\mathsf{O}}(N)$.
- *Streaming space* $= s_{\mathsf{PIOP}}(N) + s_{\mathsf{C}}(N) + s_{\mathsf{O}}(N)$.

We note that the construction underlying Lemma 4.1 enables optimizations that are not possible in the read-only setting; we detail these in Section 8. Applying Lemma 4.1 to the RW streaming PIOP and PC scheme[9] that we construct in Sections 6 and 7 leads to the following corollary:

Corollary 4.2. *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 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 7 both require $O(N)$ cryptographic time, $O(\log N)$ random-access space, and $O(N)$ streaming space. □

Remark 4.3. The foregoing discussion does not specify how RW streams for the witness are produced. In Section 9 we show how to do so for the HyperPlonk relation so that the random-access space scales with the space complexity of the computation, instead of the size of the corresponding circuit.

5 Read-write streaming sumcheck

A core component of many PIOPs (including ours and HyperPlonk's) is the sumcheck protocol [38]. In this section, we show how to construct linear-time read-write streaming provers for the sumcheck protocol when specialized to sums of products of multilinear polynomials (this is without loss of generality). 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 5.1.

Background: sumcheck. The sumcheck problem requires a prover P to convince a verifier V that $\sum_{\boldsymbol{b}\in\{0,1\}^n} f(\boldsymbol{b}) = \sigma$, where $f(\boldsymbol{X})$ is an n-variate polynomial of individual degree at most d over a field $\mathbb{F}$, and $\sigma \in \mathbb{F}$ is a claimed sum. The sumcheck protocol [38] is a protocol for this problem.

PIOP 1: Sumcheck

$\langle \mathsf{P}(f), \mathsf{V}([\![f]\!], d, \sigma)\rangle$:
For i in $[n,\ldots,1]$:
1. P sends to V the following univariate degree d polynomial: $a_i(X_i) := \sum_{\boldsymbol{b}\in\{0,1\}^{i-1}} f(\boldsymbol{b}, X_i, r_{i+1},\ldots,r_n)$.
2. V checks that a_i is of degree at most d.
3. If $i = n$, V sets $\sigma_i := \sigma$; else it sets $\sigma_i := a_{i-1}(r_{i-1})$.
4. V checks if $a_i(0) + a_i(1) = \sigma_i$.
5. V samples $r_i \xleftarrow{\$} \mathbb{F}$.
6. If $i = 1$, V asserts $a_1(r_1) = f(r_1,\ldots,r_n)$; else it sends r_i to P.

Like prior sumcheck-based SNARKs [47, 24], we assume that f is a sum of products of at most d-many multilinear polynomials $p_1,\ldots,p_d$. That is, $f := h(p_1,\ldots,p_d)$, where h is a multilinear polynomial with ℓ monomials. We refer to polynomials f of this form as (d,ℓ)-sumprod polynomials.

P's input is represented by $\boldsymbol{p}_1,\ldots,\boldsymbol{p}_d$: the evaluations of the multilinear polynomials $p_1,\ldots,p_d$ over the boolean hypercube $\{0,1\}^n$ ordered lexicographically. Below we recap Thaler's [49, 55] linear-time prover for the sumcheck PIOP for product of d multilinear polynomials (so $\ell = 1$), as it will be the basis for our RW streaming sumcheck prover. We omit the details of the work of V since they are unchanged. At a high level, in each round Thaler's algorithm first interpolates the polynomial a_i (Step 4 and the Sum helper function), and then eliminates the i-th variable via the FoldInPlace helper.

$\mathsf{P}(\boldsymbol{p}_1,\ldots,\boldsymbol{p}_d)$:
1. Initialize table $\boldsymbol{T}_j \leftarrow \boldsymbol{p}_j$ for all $j \in [d]$.
2. For i in $[n,\ldots,1]$:
3. Define $N_i := N/2^{n-i}$.
4. Set $S \leftarrow \mathsf{Sum}(N_i, d, \boldsymbol{T}_1,\ldots,\boldsymbol{T}_d)$.
5. Send $[a_i(s) := S[s]]_{s=0}^{d}$ to V.
6. If $i \neq 1$:
7. Receive challenge $r_i \xleftarrow{\$} \mathbb{F}$ from V.
8. $\mathsf{FoldInPlace}(N_i, 1-r_i, r_i, \boldsymbol{T}_j)$ for all $j \in [d]$.

$\mathsf{Sum}(N, d, \boldsymbol{T}_1,\ldots,\boldsymbol{T}_d) \to S$:
1. Set $S \leftarrow [0]_{s=0}^{d}$.
2. For j in $[1,\ldots,N/2]$ and s in $[0,\ldots,d]$:
3. $S[s] \leftarrow S[s] + \prod_{i=1}^{d} ((1-s)\cdot T_i[2j] + s\cdot T_i[2j+1])$.

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

The foregoing algorithm requires a table of size $N = 2^n$ for each multilinear polynomial, 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 $O(d\cdot N)$, but also, unfortunately, to a space complexity of $O(d\cdot N)$.

[9] 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.

5.1 Linear-time RW streaming sumcheck

Prior work indicates that algorithms for sumcheck with random-access space $O(\text{polylog } N)$ must take time $\Omega(N \log N / \log\log N)$ [5]. We design our RW streaming algorithm to avoid this lower bound.

Our algorithm. Notice that in each round of Thaler's algorithm, all operations except Sum and FoldInPlace require $O(1)$ time and space. So to construct a RW streaming version of Thaler's prover algorithm, it suffices to construct RW streaming algorithms for these. We describe the result below.

Read-write Streaming Algorithm 1: SUMCHECK for (d,ℓ)-sumprod polynomials

$\mathsf{P}(\mathbf{R}_1,\ldots,\mathbf{R}_d)$:
1. Initialize tables $\boldsymbol{T}_i \leftarrow \mathbf{R}_i$.
2. For each i in $[n,\ldots,1]$:
3. Initialize running sums: $S \leftarrow [0]_{j=0}^{d}$.
4. For each monomial $(c \cdot X_{j_1} \cdot X_{j_2} \cdot \ldots \cdot X_{j_k})$ in h:
5. Obtain $d+1$ evaluations for the round polynomial of the monomial: set $E \leftarrow \mathsf{RWSum}(d, \boldsymbol{T}_{j_1},\ldots,\boldsymbol{T}_{j_k})$.
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: $\boldsymbol{T}_{j_1}.\mathsf{restart}(),\ldots,\boldsymbol{T}_{j_k}.\mathsf{restart}()$.
8. Send $[a_i(s) := S[s]]_{s=0}^{d}$ to V.
9. If $i \neq 1$:
10. Receive verifier challenge $r_i \xleftarrow{\$} \mathbb{F}$ from V.
11. Fold streams: $\mathsf{RWFoldInPlace}(1-r_i, r_i, \boldsymbol{T}_1,\ldots,\boldsymbol{T}_d)$.

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

$\mathsf{RWFoldInPlace}(\alpha, \beta, \boldsymbol{T}_1,\ldots,\boldsymbol{T}_d)$:
1. Allocate intermediate RW stream: $\mathbf{W}.\mathsf{init}(\boldsymbol{T}_i.\mathsf{len}()/2)$.
2. For i in $[1,\ldots,d]$:
3. Set $\mathbf{W}_i \leftarrow \mathsf{Map}((a,b)\mapsto \alpha\cdot a + \beta \cdot b) \rightsquigarrow \mathsf{Zip} \rightsquigarrow \mathsf{SplitEO} \rightsquigarrow \boldsymbol{T}_i$.
4. Set $\boldsymbol{T}_i \leftarrow \mathbf{W}_i.\mathsf{swapmode}()$.

It is straightforward to see that this algorithm preserves the desired time complexity of $O(d \cdot \ell \cdot N)$ and reduces the random-access space complexity to just $O(d+\ell)$.

Optimizations. If two monomials in h share a common term p_j, we do not need to make two passes over $\mathbf{R}_j$ to compute the monomials' contributions to the round polynomial. Instead, we can read from $\mathbf{R}_j$ once and store the value in RAM. This value can be used to compute both contributions. This enables a single pass over each $\mathbf{R}_j$, as opposed to worst-case ℓ passes. Additionally, if one of the input polynomials is eq, then P can generate its evaluations in a *read-only* streaming manner in $O(N)$ time and $O(\log N)$ space, thus reducing I/O costs. See Section 6.1 and Appendix B for details.

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.

6.1 Zerocheck

Given a polynomial p, the zerocheck PIOP [24] 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 $\mathrm{eq}(\boldsymbol{r},\boldsymbol{b}) = \prod_{i=1}^{n}((1-r_i)(1-b_i) + r_i b_i)$.)

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.[10] 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 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 evaluations over the entire hypercube would require $O(N \log N)$ time. We achieve $O(N)$ time by fleshing out an observation from an unpublished manuscript of Vu in 2013. The resulting algorithm proceeds as follows. First, it invokes an initialization procedure $\mathsf{Eq.Init}(\boldsymbol{r})$ that, on input $\boldsymbol{r}$, initializes a state in $O(\log N)$ time. Then, to generate the stream $[\mathrm{eq}(\boldsymbol{r},\boldsymbol{b})]_{\boldsymbol{b}\in\{0,1\}^n}$, it sequentially invokes $\mathsf{Eq.Next}()$, 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 $\boldsymbol{r}$.

[10] We provide a concrete optimization of this reduction via the *Lagrange sumcheck protocol* in Appendix B.

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 $\mathsf{prod} \leftarrow \mathsf{prod} \cdot r_j/(1 - r_j)$; $\mathsf{out} \leftarrow \mathsf{prod}$; break.
7. Else, set $\mathsf{out} \leftarrow 0$. // (there exists $i \in S$ s.t. $r_i \neq b_i$)
8. Increment (the integer represented by) $\boldsymbol{b}$. // ($O(1)$ time)
9. Output out.

We first analyze the case where $\boldsymbol{r} \in (\mathbb{F} \setminus \{0,1\})^n$, implying that $S = \varnothing$. 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 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 $\boldsymbol{b}$.

While a single call to Eq.Next can, in the worst case, cost $O(n) = O(\log N)$ time, but over N invocations, the cost amortizes to $O(1)$ per invocation. 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 (and hence the total number of multiplications) required to compute the evaluations of eq is at most $2N$. 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 either r_i or by $(1 - r_i)$, 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.

Remark 6.1 (comparison to Vu's procedure)**.** Vu gives a high level description of the foregoing algorithm, but this description omits numerous details, and in particular does not handle the case where some of the r_i's are boolean. The latter is important for supporting batch opening of multiple polynomials at different points [24].

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 [48]. In this PIOP, the prover P computes and sends to the verifier an $(n+1)$-variate polynomial ν such that $\nu(0, \boldsymbol{X}) := p(\boldsymbol{X})/q(\boldsymbol{X})$ and $\nu(1, \boldsymbol{X}) := \nu(\boldsymbol{X}, 0) \cdot \nu(\boldsymbol{X}, 1)$. 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 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 $O(N)$ space, this requires $O(N)$ time. What about with less 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 this is straightforward for $\nu^{(0)}$, 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. As the latter is not possible in the low-memory regime, we must use the first approach which requires $O(N \log N)$ time overall.

Remark 6.2. An alternative approach 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 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.

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 Appendix C a detailed description of this algorithm and the overall PIOP.

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: "$\{\{p(\boldsymbol{x}) : \boldsymbol{x} \in \{0,1\}^n\}\} = \{\{q(\boldsymbol{x}) : \boldsymbol{x} \in \{0,1\}^n\}\}$".[11]

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 $\prod_{\boldsymbol{x} \in \{0,1\}^n} (p(\boldsymbol{x}) + \alpha)/(q(\boldsymbol{x}) + \alpha) = 1$.

[11] The double braces notation $\{\{\cdot\}\}$ represents multisets.

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).

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 proves that "$p(\boldsymbol{x}) = q(\pi(\boldsymbol{x}))$ for all $\boldsymbol{x} \in \{0,1\}^n$".

Reduction to multiset-equality-check. This claim is proven via 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 $\{p(\boldsymbol{x})+\beta\cdot \mathsf{int}(\boldsymbol{x})\}_{\boldsymbol{x}\in\{0,1\}^n} = \{q(\boldsymbol{x})+\beta\cdot \mathsf{int}(\pi(\boldsymbol{x}))\}_{\boldsymbol{x}\in\{0,1\}^n}$.[12]

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 produced in $O(N)$ time, and only $O(1)$ additional space (both streaming and random-access).

6.5 PIOP for HyperPlonk

Recall from Section 2.1 that HyperPlonk's PIOP reduces circuit satisfiability to a permcheck claim and a zerocheck claim. We show in Appendix C how to construct a RW streaming prover for this PIOP that requires $O(N)$ time, $O(\log N)$ random-access space, and $O(N)$ streaming space complexity and satisfies Corollary 4.2. The time and space complexities of the aforementioned PIOPs are summarized in Table 1.

7 Read-write streaming PC schemes

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

We now provide an overview of our construction of read-write streaming algorithms for the PST scheme [42], and defer to Appendix D an overview of our other constructions.

[12] 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 $\mathbb{F}$.

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. Verifier time, query complexity, and soundness error are the same as their HyperPlonk counterparts [24].

7.1 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. For a group of size $O(2^\lambda)$, this requires N scalar multiplications. Since each of the latter requires $O(\lambda)$ group additions, the overall cost is $O(\lambda N)$ group additions.

Fast read-only streaming MSMs. Pippenger's algorithm [45] computes an MSM of size N in a group $\mathbb{G}$ of size $O(2^\lambda)$ with just $O(\lambda \cdot N / \log(\lambda \cdot N))$ group additions. We can apply this algorithm "in chunks" to construct a read-only streaming MSM algorithm that uses m random-access space and $O(\lambda N / \log(\lambda m))$ group additions as follows: define a running MSM output $\mathsf{ans} := 0$. In a streaming manner, read m field and group elements of the input at a time, compute their MSM using Pippenger's algorithm, and add the output to ans. After N/m iterations, output ans.

7.2 Read-write streaming PST

The PST scheme [42] generalizes the KZG scheme [34] to multilinear polynomials. We recall Libra's [55] version of this scheme, and then describe how to make it RW streaming.

Setup. Sample 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 output the commitment key $\mathsf{ck} := ([\mathsf{ck}_j]_{j=0}^{n-1})$, where $\mathsf{ck}_j := [\mathsf{eq}_j(\boldsymbol{\alpha}_{[n-j+1:n-1]}, \boldsymbol{i}) \cdot G]_{\boldsymbol{i}\in\{0,1\}^j}$, and $\boldsymbol{\alpha} \xleftarrow{\$} \mathbb{F}^n$ is uniformly random. Each ck_j enables committing to a j-variate multilinear polynomial.[13]

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

scheme	SRS size	time		streaming space		check time	proof size
		commit	open	commit	open		
PST13 [42]	$O(N)$	$O(N)$	$O(N)$	$O(1)$	$O(N)$	$O(\log N)$	$O(\log N)$
Hyrax [51]	$O(1)$	$O(N)$	$O(N)$	$O(1)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N})$
Multilinear Halo [18, 47]	$O(1)$	$O(N)$	$O(N)$	$O(1)$	$O(N)$	$O(N)$	$O(\log N)$
BMMTV21 [23]	$O(\sqrt{N})$	$O(N)$	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\log N)$	$O(\log N)$
Dory [37]	$O(\sqrt{N})$	$O(N)$	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\log N)$	$O(\log N)$

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.

Commit. Given the evaluations $\boldsymbol{p}$ of an n-variate multilinear polynomial p, compute $C := \langle \boldsymbol{p}, \mathsf{ck}_n \rangle$ via a streaming MSM.

Opening. 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 complexity of computing the witness polynomials.

Zhang et al. [57, Appendix G] demonstrate a linear-time algorithm for computing these polynomials that relies on the following decomposition of the multilinear polynomial p:

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

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. [57] show how to compute this decomposition in linear time.

Limitations of read-only streaming. It is straightforward to construct a read-only streaming algorithm for Commit, but doing the same for Open is more difficult. A naive 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 from scratch and in linear time, the evaluations $\boldsymbol{q}_i$ of q_i.

We can avoid this by changing the order in which we produce $\boldsymbol{q}_i$ (similar to our streaming prodcheck algorithm in Section 6.2). 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. traverses this tree in a breadth-first manner, computing all the evaluations of a particular $\boldsymbol{q}_i$ before moving on to the next one. Traversing this tree in depth-first manner would reduce this cost to $O(N)$ time overall. Unfortunately, this approach produces evaluations of the q_i's in an interleaved manner (e.g. $q_1[0], q_2[0], \cdots$), and committing to these in a read-only streaming manner requires a similarly interleaved commitment key, thus doubling overall commitment key size.

RW streaming in cryptographic linear-time. We describe a simpler, cryptographic linear-time, RW streaming Open algorithm that avoids interleaving. Our construction follows from the observation that reversing the order of variable elimination in the algorithm of Zhang et al. enables one to easily produce streams for $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n$; indeed, the algorithm resembles RWFoldInPlace from Section 5.1.

Let $\boldsymbol{p}$, $\boldsymbol{g}$ and $\boldsymbol{h}$ be the evaluations (over the appropriate hypercubes) of $p(X_1, \ldots, X_n)$, $g(X_1, \ldots, X_{n-1})$ and $h(X_1, \ldots, X_{n-1})$ respectively. Let $\boldsymbol{i}$ be a point in $\{0,1\}^n$. When $i_n = 0$, $p(i_1, \ldots, i_{n-1}, 0) = g(i_1, \ldots, i_{n-1})$, and when $i_n = 1$, $p(i_1, \ldots, i_{n-1}, 1) = g(i_1, \ldots, i_{n-1}) + h(i_1, \ldots, i_{n-1})$, implying that $h(i_1, \ldots, i_{n-1}) = p(i_1, \ldots, i_{n-1}, 1) - p(i_1, \ldots, i_{n-1}, 0)$, and so we can set $\boldsymbol{g} := \boldsymbol{p}_E$ and $\boldsymbol{h} := \boldsymbol{p}_O - \boldsymbol{g}$, where $\boldsymbol{p}_E$ and $\boldsymbol{p}_O$ are the even and odd halves of $\boldsymbol{p}$, respectively.

Given streaming access to $\boldsymbol{p}$, the prover writes $\boldsymbol{g}$ and $\boldsymbol{h}$ onto intermediate write streams via the foregoing identities. The prover then sets $\boldsymbol{q}_1 := \boldsymbol{h}$ and commits to it. Finally, the prover computes $\boldsymbol{r}_1 := \boldsymbol{g} + z_1 \cdot \boldsymbol{h}$ using the streams for $\boldsymbol{g}$ and $\boldsymbol{h}$ and uses it to compute $\boldsymbol{q}_2, \ldots, \boldsymbol{q}_n$.

We further reduce I/O via the following **optimizations**:

- *Batch commitments:* Often, the prover needs to commit to multiple polynomials at once. Calling Commit on each polynomial separately would incur a fresh pass over the commitment key per polynomial. We instead provide a BatchCommit algorithm that performs a single pass over the commitment key, using the group element it has read to update the commitment for each polynomial in the batch.
- *Reduced I/O for* Open*:* Instead of producing intermediate streams for $\boldsymbol{g}$ and $\boldsymbol{h}$, Open can directly produce a stream for $\boldsymbol{r}_1$ and the commitment to $\boldsymbol{q}_1$: it reads two elements of $\boldsymbol{p}$ at

a time, and first combines these to produce an element of $\boldsymbol{r}_1$, and then uses the first element to update $\boldsymbol{q}_1$'s commitment.

8 Related work

8.1 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 RW streams. Below, we discuss these works that construct such streaming SNARKs, and remark that they sacrifice on either prover or verifier time.

Comparison of frameworks. RW streaming enables more efficient composition of streaming algorithms (see Section 3.1) as the output of subroutines can be stored in external memory and reused later. This enables better concrete efficiency for RW streaming SNARKs compared to read-only streaming SNARKs: the SNARK prover $\mathcal{P}$ must feed the polynomials output by the PIOP prover to both PC.Commit and PC.Open. In the RW setting, $\mathcal{P}$ can write these polynomials to external memory and then feed the input to both PC algorithms; in the read-only setting $\mathcal{P}$ must recompute them.

Block et al. [11, 12] construct read-only streaming SNARKs by designing a streaming PIOP and combining it with streaming PC schemes from inner-product arguments [11] or groups of unknown order [12]. Both works achieve logarithmic random-access space complexity for the prover, but incur quasi-linear time complexity. They do not implement their SNARK, and so we cannot provide concrete efficiency comparisons. Our RW streaming inner-product argument is inspired by the read-only streaming one in [11], but, unlike the latter, is able to achieve cryptographic linear time complexity.

Gemini [16] designs a read-only streaming SNARK with $O(N\log^2 N)$ prover time and logarithmic random-access memory by designing and composing streaming PIOPs and streaming PC schemes. Read-write streams do not seem to help improve the efficiency of their SNARK, as their PIOP requires sparse-matrix-vector multiplication, which seems to inherently require quasi-linear time without random-access to the vector.

Epistle [56] 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 HyperPlonk [24], but it switches out HyperPlonk's prodcheck PIOP in favor of a novel construction that is more amenable to read-only streaming. Unfortunately, since their prodcheck PIOP also reduces to sumcheck, the best time complexity it can achieve remains superlinear [5]. Epistle's code is not publicly available, but we were able to obtain a private copy, and compare against it in Section 10.

Baweja et al. [5] introduce a read-only streaming sumcheck prover for multilinear polynomials which requires $O(N^{1/k})$ space and $O(kN)$ time for a tunable parameter k. Setting $k = \log N/\log\log N$ gives a $O(\log N)$ space and $O(N\log N/\log\log N)$ time sumcheck prover. They show this is optimal via an unconditional matching lower bound.

Sparrow [44] 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.

Hobbit [43] designs a sumcheck protocol for products of multilinear polynomials which requires $O(N)$ time and $O(\sqrt{N})$ random-access space, albeit at the cost of an increased proof size, and round and verifier complexity of $O(\sqrt{N})$.

8.2 Complexity-preserving SNARKs

Complexity-preserving SNARKs [10] impose at most a polylogarithmic multiplicative overhead in prover time and space over 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 [8]. Unfortunately, IVC-based approaches generally only support uniform computations and require non-black-box use of cryptography. The latter leads to further issues such as concrete efficiency overheads in some cases, difficulties in achieving provable security, and a need for heuristics due to the reliance on random oracles, all stemming Nevertheless, IVC-based complexity-preserving SNARKs achieve good concrete efficiency.

Mangrove [40] builds a complexity-preserving SNARK by reducing circuit-satisfiability to a uniform computation, and applying efficient accumulation/folding-based IVC schemes [20, 35] 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 [52] Ligetron builds an argument of knowledge with prover time $O(N\log N)$ and $O(\sqrt{N})$ random-access space, improving upon a prior theoretical work of Bangalore et al. [4]. Unlike Scribe, Ligetron does not have a succinct verifier. Ligetron achieves good concrete efficiency.

9 Implementation

We implemented Scribe in Rust atop the `arkworks` framework [26] by adapting and extending the HyperPlonk implementation.[14] Our implementation is modular and allows for switching out almost all components, from the PC scheme to

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

the underlying elliptic curve. We also adapt the `jellyfish` circuit construction framework[15] to output witness streams for the circuits it constructs. We now describe the infrastructure we developed to enable efficient high-level implementations of read-write streaming algorithms.

9.1 Tools for read-write streams

To implement the read-write streaming algorithms outlined in Sections 6 and 7, 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). Our routines simultaneously minimize on-disk size and the overhead of (de)serialization. For example, we serialize elliptic curve points via a non-standard compression scheme [28] that compresses two points into three field elements. While compressed size is worse than in the standard method, decompression cost is much lower, leading to overall savings.

File-backed vectors. We provide a new `FileVec` vector type whose API resembles that of Rust's standard `Vec` type, but which uses temporary files on disk as backing storage for the vector. `FileVec` automatically switches to memory-backed storage when the vector becomes small enough. We engineer `FileVec`'s API to ensure sequential accesses. To ensure that these accesses do not populate the OS's file-system cache and increase memory usage, `FileVec` opens files with the `O_DIRECT` flag on Linux and the `F_NOCACHE` flag on MacOS.

File-backed multilinear polynomials. We use `FileVec` to build a multilinear polynomial type whose evaluations are either stored on disk, or can be streamed in a read-only manner when possible (e.g., the Lagrange basis polynomial).

Batched iterators. We implement the various stream operations required by our algorithms via a new *batched iterator* interface. We specify this interface as a Rust trait, `BatchedIterator`, whose API resembles the standard `Iterator` trait, but which processes (in parallel via `rayon`[16]) 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`.

10 Evaluation

We evalauted Scribe's performance in a variety of settings, with the aim of answering the following questions:

- **Q1:** Can Scribe scale to prove large circuits, and how does time and disk space usage scale with circuit size?
- **Q2:** What is the overhead over memory-intensive proving as the compute-I/O ratio varies?
- **Q3:** What is the memory cost of witness synthesis?

Baselines. We compare Scribe against the following baselines: memory-intensive HyperPlonk (HP) [24] and low-memory Gemini [16] and Epistle [56]. We chose these baselines since they operate in the same regime as Scribe: low proof size, and support for general circuits. Both Scribe and HP use PST as the PC scheme. Other low-memory SNARKs discussed in Section 8 fail to meet one or more of these criteria. All baselines use `arkworks` v0.4, while Scribe uses v0.5. Running `arkworks`'s benchmark suite on both versions found minimal performance differences; see Appendix A for details.

We also compared against state-of-the-art industrial proof systems Halo2[17] and Plonky2[18], but they could not scale to large instances, and so we omitted comparisons against them.

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 $\leq 2^{17}$ elements.

Experimental setups. We ran our experiments on an AWS EC2 `i3en.3xlarge` 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/hr at the time of writing. We enforce memory and bandwidth limits via Linux's `cgroups` functionality. Scribe, Gemini, and Epistle are configured to use at most 750 MiB of memory, while HP is allowed to use all available memory.

To test Scribe's performance on a low-memory mobile device, we also ran some experiments on an iPhone 13 Pro Max with 6 cores, 6 GiB RAM, and 256 GiB of disk storage.

10.1 Q1: Scalability

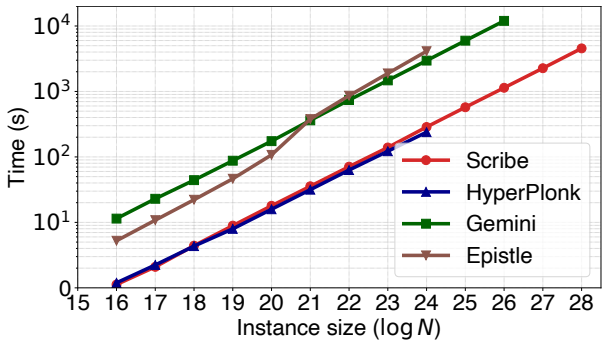

Figure 2: Proving time on AWS.

We evaluated Scribe's resource usage in both experimental setups when scaling to large instances. We focus below on

[15] https://github.com/EspressoSystems/jellyfish
[16] https://github.com/rayon-rs/rayon
[17] https://github.com/privacy-scaling-explorations/halo2
[18] https://github.com/0xPolygonZero/plonky2

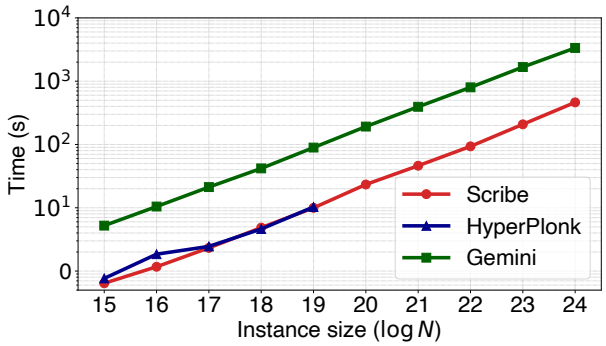

Figure 3: Proving time on iPhone.

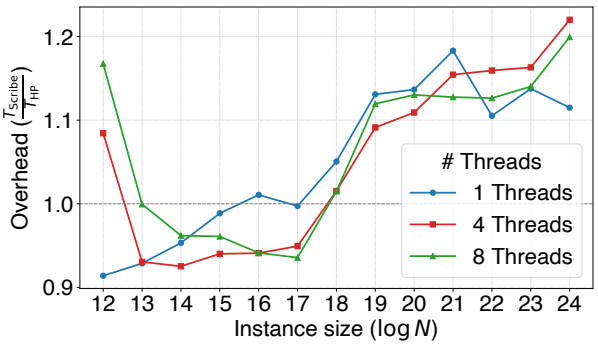

Figure 4: Overhead of Scribe as number of threads varies.

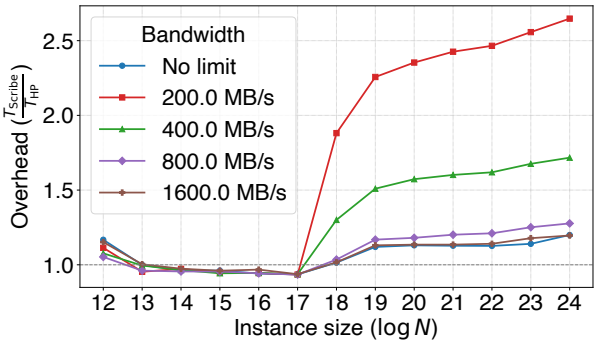

Figure 5: Overhead of Scribe when bandwidth is limited.

the AWS setup, as the iPhone results are qualitatively similar. The takeaway is that Scribe can prove large instances with low time and disk usage overheads.

Q1a: running time. Fig. 2 shows that Scribe's latency scales linearly with instance size. This latency is 10.5× smaller than Gemini's, 13.5× smaller than Epistle's, and is just 1.1-1.2× larger than HP's latency. Unlike Epistle, Scribe does not incur an increase in proving time when moving from in-memory to streaming proving.

Q1b: disk space. Scribe's disk usage scales linearly with instance size, and grows at roughly 480 bytes per gate. At the largest instance size (2^{28}), Scribe used 145 GB of disk.

Proving time on iPhone. Fig. 3 shows that Scribe's latency on the iPhone is 7× smaller than Gemini's, and is just 1.1-1.2× larger than HP's latency. Scribe can scale to 2^{24} gates on the iPhone, while HP exhausts system memory at 2^{19} gates (32× fewer). We omit a comparison with Epistle as it is slower than Gemini. We omit measurements of disk usage as these are similar to the AWS setup.

10.2 Q2: Overhead of read-write-streaming

We investigate the overhead of read-write streaming in Scribe via experiments which vary compute-to-I/O ratios.

Faster compute via more threads. The first experiment varies the number of threads used by Scribe and HP and is reported in Fig. 4. The latter shows that Scribe's overhead does not vary significantly with the number of threads, and so, for a reasonable number of threads, Scribe is not I/O bound.

Slower I/O via bandwidth throttling. Our second experiment fixes the number of threads to 8 and instead varies disk bandwidth. Fig. 5 shows that when throttling bandwidth degrades Scribe's latency, indicating an I/O bottleneck. However, even this degraded latency is $< 3\times$ that of HP.

10.3 Cost of witness synthesis

To evaluate the time and memory costs of witness synthesis (converting high-level witnesses to wire/variable assignments) in our circuit programming framework, we sample randomly-generated circuits and 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, and a *replacement probability* that specifies the probability that a variable in S is replaced by a new variable. It produces a circuit whose every gate is randomly sampled to be either an add or a multiply gate, and each gate's inputs are randomly sampled from S. 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 sizes for S. The following table shows that these costs are a miniscule fraction of proof generation costs.

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

Ethical Considerations

This work develops a new memory-efficient SNARK construction and a model that can be used to describe and analyze such constructions.

The research did not involve human subjects, personal data, or experiments on live systems, so it avoids many of the direct risks common in security research. The main stakeholders are cryptography researchers, protocol developers, and end users who may rely on systems that adopt our work.

The impact of our research on the first two parties is mainly in providing them with increased flexibility in designing their applications and systems: Scribe provides them with a new tool that can improve the efficiency of their applications.

Analysing the impact on end users requires more nuance. A key application of Scribe is in privacy-preserving blockchains where client devices produce SNARK proofs (e.g., private transaction and private smart contract blockchains). The efficiency improvements provided by Scribe can increase the adoption of such privacy-preserving systems by enabling low-powered client devices to efficiently produce proofs. This has both benefits and downsides.

On the one hand, users of existing non-private blockchain systems have essentially no privacy guarantees, so enabling them to easily use private blockchains can greatly increase their financial and data privacy. On the other hand, private blockchains can also be abused for less-than-ideal purposes, such as difficult-to-detect money laundering. This kind of trade-off is not unique to our system, and is a long-standing question faced by developers of other privacy-preserving technology, such as end-to-end-encrypted messengers. In our opinion, the benefits of improved privacy are worth the downsides. To maximize the chances that these benefits are realized, both the research paper and the associated software artifacts and our code are publicly available.

Open Science

All artifacts needed to evaluate our work are available at `https://zenodo.org/records/17957696`. The repository is structured as a Rust workspace. It contains an implementation of our streaming infrastructure, an implementation of our Scribe, and benchmark harnesses used for all performance measurements.

We note that while our paper compares with Epistle, our benchmark harness does not support automated experiments with Epistle as its code is not publicly available; our benchmarks rely on a private copy of Epistle's code that we obtained from the authors.

The repository includes build instructions and scripts to run experiments on different parameter sizes. This repository is fully self-contained and allows reviewers to check correctness and performance without relying on external resources.

Acknowledgements

Pratyush Mishra and Matan Shtepel are partially supported by a Sui Academic Research Award. Pratyush Mishra and Tushar Mopuri are partially supported by a Sony Research Award. Pratyush Mishra is additionally supported by a gift from Ingonyama.

References

[1] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo. "Structure-Preserving Signatures and Commitments to Group Elements". In: *J. Cryptol.* (2016).

[2] A. Aggarwal and J. S. Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Communications of the ACM* (1988).

[3] A. Arun, S. T. V. Setty, and J. Thaler. "Jolt: SNARKs for Virtual Machines via Lookups". In: EUROCRYPT '24.

[4] 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: TCC '22.

[5] A. Baweja, A. Chiesa, E. Fedele, G. Fenzi, P. Mishra, T. Mopuri, and A. Zitek-Estrada. "Time-Space Trade-Offs for Sumcheck". In: TCC '25.

[6] P. Beame, T. S. Jayram, and A. Rudra. "Lower Bounds for Randomized Read/Write Stream Algorithms". In: STOC '07.

[7] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves". In: CRYPTO '14.

[8] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. "From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again". In: ITCS '12.

[9] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. "Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data". In: STOC '13.

[10] N. Bitansky and A. Chiesa. "Succinct Arguments from Multi-Prover Interactive Proofs and their Efficiency Benefits". In: CRYPTO '12.

[11] 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: TCC '20.

[12] A. R. Block, J. Holmgren, A. Rosen, R. D. Rothblum, and P. Soni. "Time- and Space-Efficient Arguments from Groups of Unknown Order". In: CRYPTO '21.

[13] D. Boneh, J. Drake, B. Fisch, and A. Gabizon. "Halo Infinite: Proof-Carrying Data from Additive Polynomial Commitments". In: CRYPTO '21.

[14] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. "Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting". In: EUROCRYPT '16.

[15] J. Bootle, A. Cerulli, J. Groth, S. K. Jakobsen, and M. Maller. "Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution". In: ASIACRYPT '18.

[16] J. Bootle, A. Chiesa, Y. Hu, and M. Orrù. "Gemini: Elastic SNARKs for Diverse Environments". In: EUROCRYPT '22.

[17] J. Bootle, A. Chiesa, and K. Sotiraki. "Sumcheck Arguments and Their Applications". In: CRYPTO '21.

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

[19] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: S&P '18.

[20] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner. "Proof-Carrying Data Without Succinct Arguments". In: CRYPTO '21.

[21] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. "Proof-Carrying Data from Accumulation Schemes". In: TCC '20.

[22] B. Bünz, B. Fisch, and A. Szepieniec. "Transparent SNARKs from DARK Compilers". In: EUROCRYPT '20.

[23] B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. "Proofs for Inner Pairing Products and Applications". In: ASIACRYPT '21.

[24] B. Chen, B. Bünz, D. Boneh, and Z. Zhang. "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates". In: EUROCRYPT '23.

[25] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: EUROCRYPT '20.

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

[27] J. Cook and D. Moshkovitz. "Explicit Time and Space Efficient Encoders Exist Only with Random Access". In: CCC '24.

[28] X. Fan, A. Otemissov, F. Sica, and A. Sidorenko. "Multiple point compression on elliptic curves". In: *Des. Codes Cryptogr.* (2017).

[29] A. Fiat and A. Shamir. "How to prove yourself: practical solutions to identification and signature problems". In: CRYPTO '86.

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

[31] A. Golovnev, J. Lee, S. T. V. Setty, J. Thaler, and R. S. Wahby. "Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS". In: CRYPTO '23.

[32] M. Grohe, C. Koch, and N. Schweikardt. "Tight Lower Bounds for Query Processing on Streaming and External Memory Data". In: ICALP '05.

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

[34] A. Kate, G. M. Zaverucha, and I. Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: ASIACRYPT '10.

[35] A. Kothapalli, S. T. V. Setty, and I. Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: CRYPTO '22.

[36] R. W. F. Lai, G. Malavolta, and V. Ronge. "Succinct Arguments for Bilinear Group Arithmetic: Practical Structure-Preserving Cryptography". In: CCS '19.

[37] J. Lee. "Dory: Efficient, Transparent Arguments for Generalised Inner Products and Polynomial Commitments". In: TCC '21.

[38] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. "Algebraic Methods for Interactive Proof Systems". In: *JACM* (1992).

[39] V. Nair, J. Thaler, and M. Zhu. "Proving CPU Executions in Small Space". IACR ePrint Report 2025/611.

[40] W. Nguyen, T. Datta, B. Chen, N. Tyagi, and D. Boneh. "Mangrove: A Scalable Framework for Folding-based SNARKs". In: CRYPTO '24.

[41] P. A. Papakonstantinou and G. Yang. "Cryptography with Streaming Algorithms". In: CRYPTO '14.

[42] C. Papamanthou, E. Shi, and R. Tamassia. "Signatures of Correct Computation". In: TCC '13.

[43] C. Pappas and D. Papadopoulos. "Hobbit: Space-Efficient zkSNARK with Optimal Prover Time". In: USENIX Security '25.

[44] C. Pappas and D. Papadopoulos. "Sparrow: Space-Efficient zkSNARK for Data-Parallel Circuits and Applications to Zero-Knowledge Decision Trees". In: CCS '24.

[45] N. Pippenger. "On the Evaluation of Powers and Monomials". In: *SIAM J. Comp.* (1980).

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

[47] S. Setty. "Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup". In: CRYPTO '20.

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

[49] J. Thaler. "Time-Optimal Interactive Proofs for Circuit Evaluation". In: CRYPTO '13.

[50] J. S. Vitter and E. A. M. Shriver. "Algorithms for Parallel Memory, I: Two-level Memories". In: *Algorithmica* (1994).

[51] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. "Doubly-Efficient zkSNARKs Without Trusted Setup". In: IEEE S&P '18.

[52] R. Wang, C. Hazay, and M. Venkitasubramaniam. "Ligetron: Lightweight Scalable End-to-End Zero-Knowledge Proofs. Post-Quantum ZK-SNARKs on a Browser". In: IEEE S&P '24.

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

[54] 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: CCS '22.

[55] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation". In: CRYPTO '19.

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

[57] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vRAM: Faster Verifiable RAM with Program-Independent Preprocessing". In: IEEE S&P '18.

A Comparing arkworks versions

The baselines we compared Scribe against use an older version of arkworks. Specifically, while Scribe uses version 0.5, the baselines all use version 0.4. To illustrate that the performance differences reported in Section 10 do not arise because of this version difference, we ran relevant benchmarks from the benchmark suite of arkworks. We report the results in the following table. Note that all MSMs are of size 2^{16}, and the sparse MSM uses scalars of size 32 bits. The takeaway is that field arithmetic performs identically, while MSMs are actually faster in v0.4. Hence, if anything, the version difference benefits the baselines.

benchmark	v0.4	v0.5	speedup
$\mathbb{F}_r$ addition	2.5ns	2.5ns	0.99
$\mathbb{F}_r$ double	2.5ns	2.5ns	1.00
$\mathbb{F}_r$ inverse	3.5ns	3.5ns	1.00
$\mathbb{F}_r$ multiplication	12.2ns	12.5ns	0.97
$\mathbb{F}_r$ negation	2.8ns	2.7ns	1.00
$\mathbb{F}_r$ square	11.4ns	11.5ns	1.00
$\mathbb{F}_r$ subtraction	3.2ns	3.2ns	0.98
$\mathbb{G}_1$ MSM	844.5ms	975.4ms	0.87
$\mathbb{G}_1$ sparse MSM	140.6ms	155.8ms	0.90

B Lagrange sumcheck

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 $\boldsymbol{t}$).

Given n-variate polynomials $p_1,\ldots,p_d$, a multilinear polynomial h with ℓ monomials, a target sum $\sigma\in\mathbb{F}$, and a vector $\boldsymbol{t}\in\mathbb{F}^n$, they satisfy the Lagrange sumprod sumcheck relation if $\sum_{\boldsymbol{x}\in\{0,1\}^n} h(p_1(\boldsymbol{x}),\ldots,p_d(\boldsymbol{x}))\cdot\mathrm{eq}(\boldsymbol{t},\boldsymbol{x})=\sigma$.

The protocol is similar to Algorithm 1, except the polynomial $\mathrm{eq}(\boldsymbol{t},\boldsymbol{X})$ is never stored in streaming memory and its folded evaluations are computed on-the-fly. This protocol significantly improves the concrete efficiency of the RW streaming zerocheck PIOP.

Read-write Streaming Algorithm 2:
Lagrange sumcheck for sumprod polynomials

$\mathsf{P}(\mathbf{R}_1(p_1),\ldots,\mathbf{R}_d(p_d))$:
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. Set $E\leftarrow\mathsf{RWSumEq}(d+1,i,\boldsymbol{t},\mathbf{R}_{j_1},\ldots,\mathbf{R}_{j_k})$.
6. For s in $[0,\ldots,d+1]$: set $S[s]\leftarrow S[s]+c\cdot E[s]$.
7. $\mathbf{R}_{j_1}.\mathsf{restart}(),\ldots,\mathbf{R}_{j_k}.\mathsf{restart}()$.
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. $\mathsf{RWFoldInPlace}(1-r_i,r_i,\mathbf{R}_1,\ldots,\mathbf{R}_d)$.
12. Obtain folded Lagrange polynomial by updating folding coefficient: $\gamma\leftarrow\gamma\cdot(t_i\cdot r_i+(1-t_i)\cdot(1-r_i))$.

$\mathsf{RWSumEq}(d,i,\boldsymbol{t},\mathbf{R}_1,\ldots,\mathbf{R}_k)\mapsto S$:
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. $S[s]\leftarrow S[s]+\prod_{j=1}^{k+1}\left((1-s)\cdot a_{L,j}+s\cdot a_{R,j}\right)$

Remark B.1 (Lagrange sumcheck for multilinear polynomials)**.** A practical optimization of Algorithm 2 is to use it for the special case of multilinear polynomials, i.e., when $d=1$ and $h(X)=X$. In this case, [39] show that one can construct a read-only streaming sumcheck prover that requires only $O(N)$ time and $O(\sqrt{N})$ space. This algorithm can be adapted to the RW streaming setting by using write-streams to store the intermediate tables of size $O(\sqrt{N})$ to obtain a RW streaming sumcheck prover that requires $O(N)$ time, $O(\sqrt{N})$ streaming space, and $O(\log N)$ random-access space.

C RW streaming prover for HyperPlonk

We present our formal descriptions of the RW streaming PIOPs for prodcheck, multiset-equality-check, permcheck, and the HyperPlonk relation (we omit zerocheck for (d,ℓ)-sumprod polynomials since it follows trivially from the Lagrange sumcheck PIOP). The time and space complexity of these PIOPs are summarized in Table 1, and detailed proofs of correctness can be found in the full version.

Read-write Streaming Algorithm 3: PRODCHECK

$\mathsf{P}(\mathbf{R}_p(p), \mathbf{R}_q(q))$:

Initialize RW streams for v_j as defined above:

1. Allocate space for each v_j: for j in $[0,\dots,n]$: $\mathbf{S}_{\mathsf{v},j}.\mathsf{init}(N/2^j)$.
2. $\mathbf{S}_{\mathsf{v},0} \leftarrow \mathsf{Map}(/) \rightsquigarrow \mathsf{Zip} \rightsquigarrow (\mathbf{R}_p, \mathbf{R}_q)$.

Compute each v_i using the binary tree structure:

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

Split stream of v into halves for the zerocheck instances:

8. Define $\mathsf{v}_L(\boldsymbol{X}) := \mathsf{v}(0,\boldsymbol{X})$ and $\mathsf{v}_R(\boldsymbol{X}) := \mathsf{v}(1,\boldsymbol{X})$ and $\mathsf{v}_E(\boldsymbol{X}) := \mathsf{v}(\boldsymbol{X},0)$ and $\mathsf{v}_O(\boldsymbol{X}) := \mathsf{v}(\boldsymbol{X},1)$.
9. Split v to obtain streams for $\mathsf{v}(\boldsymbol{X},0)$ and $\mathsf{v}(\boldsymbol{X},1)$: $(\mathbf{R}_{\mathsf{v},E}, \mathbf{R}_{\mathsf{v},O}) \leftarrow \mathsf{SplitEO}(\mathbf{S}_{\mathsf{v}})$.
10. Split v to obtain streams for $\mathsf{v}(0,\boldsymbol{X})$ and $\mathsf{v}(1,\boldsymbol{X})$: $(\mathbf{R}_{\mathsf{v},L}, \mathbf{R}_{\mathsf{v},R}) \leftarrow \mathsf{SplitLR}(\mathbf{S}_{\mathsf{v}})$.

11. Invoke RW streaming zerocheck PIOP for the claim "$\mathsf{v}_L(\boldsymbol{X}) \cdot q(\boldsymbol{X}) - p(\boldsymbol{X}) = 0$".
12. Invoke RW streaming zerocheck PIOP for the claim "$\mathsf{v}_E(\boldsymbol{X}) \cdot \mathsf{v}_O(\boldsymbol{X}) - \mathsf{v}_R(\boldsymbol{X}) = 0$".

Read-write Streaming Algorithm 4: MULTISET-EQUALITY-CHECK

$\mathsf{P}(\mathbf{R}_p(p), \mathbf{R}_q(q))$:

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. $\mathbf{S}_{q'} \rightsquigarrow \mathsf{Map}(x \mapsto \alpha + x) \rightsquigarrow \mathbf{R}_q$.
4. Invoke RW streaming prodcheck PIOP for the claim "$\prod_{\boldsymbol{x} \in \{0,1\}^n} p'(\boldsymbol{x})/q'(\boldsymbol{x}) = 1$".

Read-write Streaming Algorithm 5: PERMCHECK

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

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. $\mathbf{S}_{q'} \rightsquigarrow \mathsf{Map}((x,y) \mapsto x + \beta \cdot y) \rightsquigarrow \mathsf{Zip} \rightsquigarrow \left(\mathbf{R}_q, [i]_{i=1}^N\right)$.
4. Invoke RW streaming multiset-equality-check PIOP for the claim "$\{\{p'(\boldsymbol{x}) : \boldsymbol{x} \in \{0,1\}^n\}\} = \{\{q'(\boldsymbol{x}) : \boldsymbol{x} \in \{0,1\}^n\}\}$".

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

Definition C.1. *For an arithmetic circuit $C : \mathbb{F}^k \to \mathbb{F}$, the* HyperPlonk circuit representation *is a tuple $A(C) = \left(\mathbb{F}, d, N, N_p, N_w, N_q, q, \pi, f_0, \dots, f_{N_q-1}\right)$ 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,*
- *$q : \{0,\dots,N-1\} \to \{0,\dots,N_q-1\}$ is a selector function such that $q(i)$ is the "type" of the i-th gate,*
- *$\pi : \{0,\dots,N_w-1\} \times \{0,\dots,N-1\} \to \{0,\dots,N_w-1\} \times \{0,\dots,N-1\}$ 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 C.2. *The HyperPlonk relation $\mathscr{R}_{\mathsf{HPC}}$ is an indexed relation consisting of the following tuples:*

$$(\mathsf{i}_{\mathsf{HPC}}, \mathsf{x}_{\mathsf{HPC}}, \mathsf{w}_{\mathsf{HPC}}) = (A, ([\![\boldsymbol{w}]\!], \boldsymbol{p}), (\boldsymbol{w}))$$

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

Remark C.3. In Definition C.2, 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 $f_0, \dots, f_{N_q-1}$ 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 $\tilde{q}(\mathsf{bin}_{n_q}(i), \mathsf{bin}_n(j)) = 1$ if $q(j) = i$ and 0 otherwise.
- $\tilde{\pi} : \{0,1\}^{n_w+n} \to \mathbb{F}$ 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: $\tilde{f}(\boldsymbol{X}, Y_0, Y_1, \dots, Y_{N_w-1}) = f_{t(\boldsymbol{X})}(Y_0, Y_1, \dots, Y_{N_w-2}) - Y_{N_w-1}$ 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: $\hat{f}(\boldsymbol{X}) = \tilde{f}(\hat{q}(\mathsf{bin}(0), \boldsymbol{X}), \ldots, \hat{q}(\mathsf{bin}(N_q - 1), \boldsymbol{X}), \hat{w}(\mathsf{bin}(0), \boldsymbol{X}), \ldots, \hat{w}(\mathsf{bin}(N_w - 1), \boldsymbol{X}))$.

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.

Read-write Streaming Algorithm 6: HyperPlonk Relation

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

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. $(\mathbf{S}_{w,0}, \ldots, \mathbf{S}_{w,N_w-1}) \leftarrow \mathsf{SplitLSB}(\hat{w}, n_w)$.
6. Invoke the RW streaming permcheck PIOP for the claim "$\hat{w}(\boldsymbol{x}) = \hat{w}(\hat{\pi}(\boldsymbol{x}))$ for all $\boldsymbol{x} \in \{0,1\}^{n_w+n}$".
7. Invoke the RW streaming zerocheck PIOP for the claim "$\hat{f}(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$".

D Read-write streaming PC schemes from inner product arguments

A core building block for many polynomial commitment schemes [14, 51, 37, 23, 18, 21] is an inner product argument (IPA) [14, 19] or its generalization [36, 23]. 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 [23]. 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 $\mathcal{V}$.[19]

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

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 [29].

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 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 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. [11], 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

[19] 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 the full version of this paper.

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 [17] 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 5 to obtain a RW streaming GIPA prover with the same asymptotic efficiency as our direct construction.

D.1 PC schemes from inner product arguments

Evaluating a multilinear polynomial $p(\boldsymbol{X})$ at a point $\boldsymbol{z}$ is equivalent to computing the inner product $\langle \boldsymbol{p}, \mathrm{eq}_{\boldsymbol{z}} \rangle = \sum_{\boldsymbol{b} \in \{0,1\}^n} p(\boldsymbol{b}) \mathrm{eq}(\boldsymbol{z}, \boldsymbol{b})$, 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 Appendix D 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 6.1.

This is precisely the PC scheme from [18]. It requires linear verifier time and a linear-sized commitment key. We describe next RW streaming algorithms for PC schemes that avoid these drawbacks.

D.2 PC schemes from VMV products

Background: polynomials as matrices. Wahby et al. [51] 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: $M_{ij} := p_k$ for $k = i \cdot 2^m + j$, 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 $\boldsymbol{r} := [\mathrm{eq}(\boldsymbol{z}_R, \mathsf{bin}_n(i))]_{i=0}^{\sqrt{N}-1} = \otimes_{i=m+1}^{n} (1 - z_i, z_i)$.[20]

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 [51] 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 [37] and that of Bünz et al. (BMMTV21) [23], further commit to these row commitments (which are group elements) via a structure-preserving commitment scheme in groups with bilinear pairings [1] 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 the full version of this paper.

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

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

access to $\mathbf{M}$, and even the best *in-place* (i.e. non-streaming) algorithms for this task require $O(N)$ time and space.

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 $\ell_1 \cdot \boldsymbol{M}_1$.[21] 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 $\mathbf{W}$). Further details are discussed in the full version of this paper.

E Read-write streaming and external memory

The external memory model [2] and its generalization, the parallel disk model (PDM) [50], were defined to enable reasoning about the efficiency of algorithms whose state is stored in external memory. At a high level, they do so by developing a cost model where accessing a continuous block of data in external memory is much cheaper than accessing an equal number of individual locations. We recall the definition of the PDM below.

Definition E.1 ([50])**.** *A parallel data model (PDM) algorithm $\mathcal{A}$ is parameterized by an input size N, an internal memory size M, a block transfer size B, and a number of independent disks D, such that $B \ll M \ll N$. In a single I/O step, $\mathcal{A}$ can read/write a block of length B from each of the D disks simultaneously, and in a computation step it can perform a computation over the data stored in internal memory. The three main performance measures of PDM algorithms are the number of (parallel) I/O operations, the amount of internal and disk memory required for a problem of size N, and the number of computation steps (i.e., execution time).*

It is easy to see that the RW streaming model is a restricted case of the PDM, where the block transfer size is $B = 1$ and read-write accesses must be performed in a streaming manner. On the other hand, every "good" RW streaming algorithm also implies a similarly "good" PDM algorithm that simply reads a batch of items in each access (i.e., sets $B > 1$). We formalize this intuition in the following lemma.

Lemma E.2. *Let $\mathcal{A}$ be an RW streaming algorithm that on inputs of size N has time complexity $t(N)$, random-access space complexity $m(N)$, allocates at most D streams, and has streaming space complexity $s(N)$. Then, for all $B > 1$, there exists a corresponding PDM algorithm $\mathcal{B}$ with block transfer size B which, for problem size N,*

1. *uses $m(N)$ internal memory,*
2. *uses D disks with total disk memory $s(N)$,*
3. *has time complexity $t(N)$, and*
4. *has I/O complexity $\mathsf{io}(N)/B$, where $\mathsf{io}(N)$ is the number of read and write operations performed by $\mathcal{A}$.*

Proof. The PDM algorithm $\mathcal{B}$ emulates the RW streaming algorithm $\mathcal{A}$, except that it sequentially reads and writes blocks of size B instead of one element at a time. If $\mathcal{A}$ wants to read $< B$ elements from a stream, $\mathcal{B}$ still reads a full block of size B and ignores the rest of the elements. It is easy to inspect that $\mathcal{B}$ satisfies the stated efficiency claims. □

This lemma enables us to analyze our algorithms in the simpler RW streaming model.

F Restrictions of our RW streaming model

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 will occasionally let $\mathcal{A}$ initialize $\mathbf{O}$ for clarity.

Immutable input streams. Let $\mathcal{A}(\mathbf{I}) \mapsto \mathbf{O}$ be a RW streaming algorithm. To simplify analysis, it is convenient to assume that input streams $\mathbf{I}$ are immutable. On the other hand, writing pseudocode is easier when $\mathcal{A}$ can mutate the contents of $\mathbf{I}$. We resolve this issue by rewriting $\mathcal{A}$ as follows: $\mathcal{A}$ allocates an RW stream $\mathbf{S}$, copies over the contents of $\mathbf{I}$ to $\mathbf{S}$ with one pass, and uses $\mathbf{S}$ wherever it would have used $\mathbf{I}$.

Single input and output streams. For simplicity of modeling, Definition 3.2 restricts algorithms to have single input and output streams, but pseudocode is simpler without this restriction. As above, we can get the best of both worlds by automatically rewriting any algorithm with multiple input/output streams to instead have single input/output streams by concatenating the multiple streams into single ones. This does not worsen asymptotic efficiency.

[21] 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.