Jagged Polynomial Commitments (or: How to Stack Multilinears)

Tamir Hemo, Kevin Jue, Eugene Rabinovich, Gyumin Roh, and Ron D. Rothblum

Succinct*

May 21, 2025

Abstract

Modern SNARK constructions, almost ubiquitously, rely on a *polynomial commitment scheme* (PCS) — a method by which a prover can commit to a large polynomial P and later provide evaluation proofs of the form "$P(x) = y$" to the verifier.

In the context of zkVMs (i.e., proof-systems for general-purpose RAM computations), the common design is to represent the computation trace as a sequence of tables, one per CPU instruction, and commit to the these tables, or even their individual columns, as separate polynomials. Committing separately to these polynomials has a large overhead in verification costs, especially in hash-based systems.

In this work we drastically reduce this cost via a new construction which we call the *jagged PCS*. This PCS enables the prover to commit to the entire computation trace as a single polynomial, but then allows for the verifier to emulate access to the individual table or column polynomials, so that the arithmetization can proceed in the usual manner. The jagged PCS may be thought of as a sparse PCS for a very particular form of sparsity – namely, a "jagged" matrix in which each column has a different height.

Our construction of the jagged PCS is highly performant in practice. In contrast to existing sparse PCS constructions for general sparse polynomials, the jagged PCS does not require the prover to commit to any additional oracles and the prover cost is dominated by 5 finite field multiplications per element in the trace area. Furthermore, we implement the verifier as an arithmetic circuit that depends only on the total trace area — thereby significantly reducing the problem of "combinatorial explosion" often encountered in zkVM recursion.

*Emails: {tamir,kevin,eugene,min,ron}@succinct.xyz

Contents

1 Introduction

The ability to provide succinct proofs for complex general-purpose computations has moved from the domain of pure theory to an extremely important practical tool, especially in the context of scaling up blockchain technologies.

Motivated to further improve the efficiency of such proof-systems, in this work we introduce the *jagged polynomial commitment scheme (PCS)*. This scheme allows a prover and verifier to "bootstrap" any existing multilinear PCS into an extremely efficient *sparse* multilinear PCS for a particular type of sparsity that is useful in the context of the multi-table architecture common to zkVMs[1] such as SP1, Risc0, Jolt and OpenVM.

A common design architecture of modern proof-systems, especially as pertaining to zkVMs, represents the correctness of a computation via several tables, where each table corresponds to a different CPU instruction. To facilitate efficient verification, the columns of each table are thereby encoded via low degree polynomials. The instruction associated with each table determines certain constraints that all of the rows of the given table should satisfy.

Rather than having the prover send these very large column polynomials (whose total size is larger than the overall computation trace), the prover employs a *polynomial commitment scheme*. Such a PCS enables the prover to send only a small hash value representing the entire polynomial, but in such a way that a (computationally bounded) malicious prover is bound to answer evaluation queries in a manner consistent with the low degree polynomial to which it initially committed.

A caveat of this design choice is that the prover must commit to, and provide evaluation proofs of, each one of these polynomials. In the context of hash-based PCSs, on which we focus in this work, certain simple optimizations of this paradigm are known; namely, the batch FRI [BCI+23] technique. Still, as long as the prover encodes the columns separately, the verifier has to do a non-trivial amount of work that scales linearly with the number of columns in the system.

Given the fairly large number of columns involved in zkVMs, these operations actually dominate the verifier's runtime in modern hash-based systems such as SP1. Specifically, the need to answer the FRI queries to *all* of the columns becomes very substantial and also limits the number of columns that can be used in the proof-system's design.[2] Because a system such as SP1 uses a recursion circuit to reduce a linear-size proof into a constant-size one, these verifier costs translate into higher recursion costs. This fact introduces a difficult trade-off in zkVM design decisions: to reduce table areas (and hence prover costs) at the level of the "core" or "base" proof, it is often beneficial to add columns to the tables, which, on the other hand, increases the prover costs for the recursion proof.

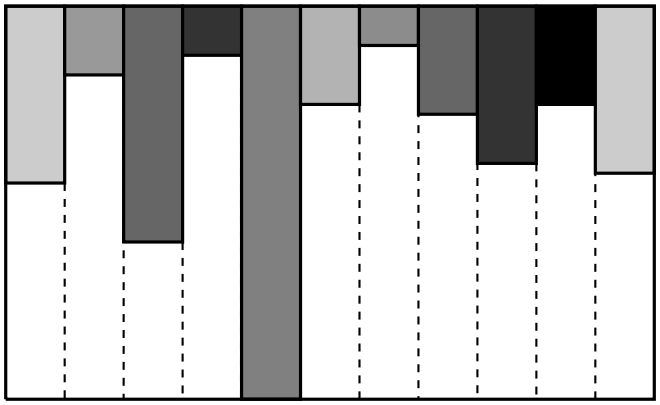
Switching from a univariate design to a multilinear one goes some way toward resolving this tension: a table of height 2^n and width 2^k can be viewed either as 2^k multilinear polynomials in m variables each, or as a single multilinear polynomial in $k+m$ variables. Hence, for the purposes of the PCS, the prover and verifier can understand the table in the latter sense, but for the purposes of proving properties of the table that hold row-by-row, they can think of it in the former sense. Then, for a multilinear PCS with logarithmic verifier costs like Basefold [ZCF24], the verifier's PCS costs are proportional to $(k+m)$ instead of $2^k \cdot m$ (if the verification protocol is run separately on each column).

[1] A zkVM is a succinct proof-system for general purpose RAM computations expressed in the instruction set of a CPU such as RISC-V.

[2] This is true even when the relevant parts of each column belong to the same leaf of a common Merkle tree.

However, there are still multiple wrinkles in the situation as described. First, we need to assume that both the height and width of the table are a power of two, which introduces significant overhead across the entire proof-system. Second, the verifier still scales linearly with the number of *tables*. Lastly, and this is perhaps most significant, the "height" 2^n of the table is not known ahead of time (e.g., for a table associated with the MULT instruction, it would correspond to the number of MULTs that were executed in the program being proved). Thus, the prover must provide this height to the verifier as part of the proof. Different values of n thus give rise to different verification circuits. In fact, in the multi-table setting of a zkVM, the prover must provide the log-height n_T for *each table* T. In the context of recursive proof composition, this causes an explosion of potential circuits that are needed in recursion. (This is a problem for zkVMs designed around univariate PCSs as well.)

Our goal is therefore to develop a "jagged adapter", which lets one commit to actual data that exists in the tables, but while enabling the verifier to emulate access to the polynomial extensions of the distinct columns. This approach enables one to use the existing arithmetization, but only pay for the cost of committing to the actual dense representation of the data (and whatever overhead the adapter requires). The different representations of the data are depicted in Fig. 1.



(a) The sparse polynomials committed to in the jagged PCS. The blank region within the dotted lines represents "virtual" zero-padding for the "actual" polynomials (the gray rectangles).

(b) The dense form of the data. A general-purpose PCS can be used to prove claims about the dense representation.

Figure 1: The sparse (a) and dense (b) representations of the data. The jagged PCS is a protocol for converting evaluation claims about (a) into claims about (b). The presence of multiple columns hints at the fact that the underlying PCS may use interleaved codes.

One way of doing this is via a *sparse PCS* which is a general-purpose protocol for reducing claims about a sparse representation of data (i.e., with a lot of zeroes) to those about a dense

representation of the same data. In that sense, a jagged PCS may be thought of as a special kind of sparse PCS, for the special case in which the data contains several tables, each of a different height. Existing general-purpose sparse PCSs such as Spark [Set20], Surge [STW24] and Spark++ [ST25] are designed for a generic pattern of sparsity and this level of generality introduces some prover overhead (see Section 1.2). In the situation outlined above, the pattern of sparsity is highly structured, and it is possible to give a dense description of the data with relatively little information in addition to the dense data itself (essentially, only the individual table heights are needed). The jagged adapter is a sparse PCS that harnesses the particular pattern of sparsity of the problem we have outlined to achieve better efficiency (see Section 1.2 for a more detailed comparison).

1.1 Our Results

In this work, the main object that we are interested in are *jagged functions*. The domain of these functions is a $2^n \times 2^k$ size table, where each of the 2^k column has a particular "height" and is zero at any point beyond this height. We formalize this notion as follows:

Definition 1.1. *Let $n, k \in \mathbb{N}$. A* jagged function with heights $\big(h_y \in [0, \ldots, 2^n - 1]\big)_{y \in \{0,1\}^k}$ *is a function $p : \{0,1\}^n \times \{0,1\}^k \to \mathbb{F}$ such that $p(x, y) = 0$ for every $x \in \{0,1\}^n$ and $y \in \{0,1\}^k$ for which $x \geq h_y$.*

Here and throughout this work we identify bit-strings such as $x \in \{0,1\}^n$ with integers in $\{0, \ldots, 2^n - 1\}$ in the natural way (i.e., via their big-endian binary representation) and freely use notation such as $x < x'$ also in the case that both are bit-strings. Also, $\mathbb{F}$ represents an arbitrary finite field that will be fixed throughout this work.

Let $n, k \in \mathbb{N}$ and let $p : \{0,1\}^n \times \{0,1\}^k \to \mathbb{F}$ be a jagged function with heights $h = \big(h_y \in [0, \ldots, 2^n - 1]\big)_{y \in \{0,1\}^k}$. We refer to the set of coordinates $\big\{(x, y) \in \{0,1\}^n \times \{0,1\}^k \text{ s.t. } x < h_y)\big\}$ (i.e., the coordinates of p which are not necessarily zero) as the *non-zero part* of p. Let $M = \sum_{y \in \{0,1\}^k} h_y$ denote the size of the non-zero part of f (where the summation is over integers). We assume wlog (i.e., by padding) that M is a power of 2, and denote by $m = \log_2(M)$.[3] We assume that $m \geq n, k$ (in typical applications m is significantly larger than n and k and this simplifies the notation).

The main goal of this work is construct a polynomial commitment scheme for the multilinear extension of p, so that the prover's work in both commitment and evaluation proof generation is $O(M)$, with very small constants. From the verifier's side, we first focus on the case that the number of columns 2^k is small (say $k \approx 10$), and so allow the verifier to run in time $O(m \cdot 2^k)$. Later we discuss strategies for reducing this cost (at a mild overhead to the prover and some additional complexity), when the number of columns is large.

Since we care about constant factors, it is crucial to specify the computational model that we use. Following both practical considerations and the norm in the literature on polynomial commitment schemes (and SNARKs more generally), we focus primarily on the number of finite field operations being done. Since, in practice, computation is usually dominated by finite field

[3] Assuming that M is a power of two can introduce a $2\times$ overhead in the worst-case. This overhead can be avoided if the commitment scheme that we use for dense polynomials supports "free padding": that is, when committing to a large multilinear whose restriction to $\{0,1\}^n$ is zero-padded, the cost of committing and generating the evaluation proof are proportional only to the non-zero part. Schemes based on interleaving such as Ligero [AHIV23], Brakedown [GLS+23] and Blaze [BCF+25] can easily be made to support such free padding.

multiplications, in order to give a precise account we count multiplications separately and refer to the other operations as addition or multiplication by small fixed constants as *basic operations*.[4]

Remark 1.2. *There are many important optimizations for PCSs that are possible when the data being committed to only contains values from a smaller base field. Examples include packing [DP25] and sumcheck optimizations [BDT24, Gru24]. To the best of our knowledge, all of these optimizations are compatible (and compounding) with the jagged PCS, but for simplicity we assume the data contains arbitrary field elements.*

The Jagged Polynomial Commitment. For every $y \in \{0,1\}^k$, let $t_y = h_1 + \cdots + h_y$, where again the summation is over the integers. In other words, the t_y's denote the *cumulative heights* up to (and including) column y. In particular, note that the t_y's are monotonically non-decreasing, and that $t_{1^k} = M = 2^m$. Note that each $t_y \leq 2^m$ and so they can be represented using m bits and indeed we view them as bit strings in $\{0,1\}^m$. We say that a sequence of cumulative heights $t = \left(t_y \in \{0,1\}^m\right)_{y \in \{0,1\}^k}$ is *admissible* if it is of the above form, namely it is non-decreasing and $t_{1^k} = 2^m$.

Fixing t as above, we construct a (natural) bijection between $\{0,1\}^m$ and the non-zero part of p via two functions $row_t : \{0,1\}^m \to \{0,1\}^n$ and $col_t : \{0,1\}^m \to \{0,1\}^k$ as follows. Given an index $i \in \{0,1\}^m$, let $y \in \{0,1\}^k$ be the smallest such that $i < t_y$. Define $col_t(i) = y$ and $row_t(i) = i - t_{y-1}$ (where for simplicity of notation we use t_{0^k-1} as a shorthand for 0).

Fact 1.3. *Fix an admissible sequence of cumulative heights t and let $p : \{0,1\}^{n+k} \to \mathbb{F}$ be a corresponding jagged function. Then, the mapping $i \mapsto (row_t(i), col_t(i))$ is a bijection between $\{0,1\}^m$ and the non-zero part of p.*

Define a function $q : \{0,1\}^m \to \mathbb{F}$ such that $q(i) = p(row_t(i), col_t(i))$, for every $i \in \{0,1\}^m$. We refer to q as the *dense* representation of p, and to p as the *sparse* representation. Recall that our goal is to efficiently commit to the multilinear extension $\hat{p} : \mathbb{F}^{n+k} \to \mathbb{F}$ of p (i.e., the unique multilinear polynomial that agrees with p on $\{0,1\}^{n+k}$). The commitment to $\hat{p}$, is the multilinear oracle $\hat{q} : \mathbb{F}^m \to \mathbb{F}$ and the cumulative heights $\{t_y\}_{y \in \{0,1\}^k}$, which are sent in the clear (i.e., not as an oracle — later we also consider methods that avoid sending the heights in the clear).

Observe that by Fact 1.3, given the polynomial $\hat{q}$ and the cumulative heights $(t_y)_{y \in \{0,1\}^k}$ the multilinear $\hat{p}$ is well defined and we denote it by $\hat{p} = sparse(\hat{q}, t)$. Our main result is a protocol which reduces a claim on the sparse representation of the function, to a claim about the dense representation.

Theorem 1.4 (Basic Jagged)**.** *There exists an m-round interactive protocol between a prover and a verifier, in which the prover gets as input a function $q : \{0,1\}^m \to \mathbb{F}$, admissible cumulative heights $t = (t_y)_{t \in \{0,1\}^k}$, a point $z \in \mathbb{F}^{n+k}$ and claimed value $v \in \mathbb{F}$. The verifier gets as input only t, z and v. The protocol satisfies the following properties:*

[4] Specifically, we assume that there is a constant sized set of field elements $\Lambda \subseteq \mathbb{F}$ so that multiplication of an arbitrary field element $\alpha \in \mathbb{F}$ with some $\lambda \in \Lambda$ counts as a basic operation (rather than a multiplication). For a prime field, this set could include small numbers such as say 2, multiplication by which is essentially an addition. Over a binary extension field, in which field elements can be represented as polynomials, the set of linear functions over the base field (viewed as extension field elements) have a similar property in the sense that multiplication by them can be easily implemented using bit shifts and additions.

1. ***Completeness:*** *for every $q : \{0,1\}^m \to \mathbb{F}$ and admissible cumulative heights t, if $\hat{p} = sparse(\hat{q}, t)$ is such that $\hat{p}(z) = v$, then when interacting with the prover, the verifier always outputs a point $z' \in \mathbb{F}^m$ and value $v' \in \mathbb{F}$ such that $\hat{q}(z') = v'$.*

2. ***Soundness:*** *for every $q : \{0,1\}^m \to \mathbb{F}$ and admissible cumulative heights t, if $\hat{p} = sparse(\hat{q}, t)$ is such that $\hat{p}(z) \neq v$, then when interacting with any prover, with probability at least $1 - \frac{2m}{|\mathbb{F}|}$ the verifier either rejects or outputs a point $z' \in \mathbb{F}^m$ and value $v' \in \mathbb{F}$ such that $\hat{q}(z') \neq v'$.*

3. ***Efficiency:*** *The prover performs $O(2^m)$ field operations, out of which at most $5 \cdot 2^m + 2^n + 2^k$ are multiplications, and the verifier is implemented as an arithmetic circuit of size $O(m \cdot 2^k)$.*

We emphasize that the verifier in Theorem 1.4 is implemented as an *arithmetic circuit*. In particular, all information about the column heights is processed purely via arithmetic over the field (rather than any conditional branching logic). This fact is important in the context of recursive proof composition, in which one proves correctness of the verifier's computation and the pure arithmetic model is far easier to prove. In fact, the circuit depends only on the log-trace-area m.

The size of the verifier in Theorem 1.4 should be contrasted to the standard solution in the literature in which the prover commits to each column separately. Even using the standard optimizations of batch FRI and of hashing each row into a single leaf of the Merkle tree, this solution involves work (specifically hashing and arithmetic) that is proportional to $\lambda \cdot 2^k$, where λ is the security parameter. In contrast, the verifier's work in Theorem 1.4 is only $O(m \cdot 2^k)$, where in practice $m \ll \lambda$. Needless to say, the constant in the big O notation. Next, in Section 1.1.1 we describe how to reduce the verification complexity to essentially $m \cdot 2^k$ multiplications (i.e., the constant becomes 1) with only a small amount of additional work for the prover.

1.1.1 Coping with a Large Number of Columns

Next, we consider the case in which the columns count 2^k is large, in which case the verifier cost in Theorem 1.4 does become prohibitive.

To show how to mitigate this cost, we need to describe the verifier's actual computation in the proof-system of Theorem 1.4. The verifier's cost in this protocol is strictly dominated by evaluating the following expression:

$$\sum_{y \in \{0,1\}^k} eq(z_c, y) \cdot \hat{g}(z_r, i, t_{y-1}, t_y), \tag{1}$$

for some fixed $z_r \in \mathbb{F}^n$, $z_c \in \mathbb{F}^k$, $i \in \mathbb{F}^m$, where $\hat{g}$ is the multilinear extension of a specific simple function $g : \{0,1\}^{n+3m} \to \{0,1\}$, the t_y's are the cumulative column heights and where, for simplicity of the notation, we set t_{0^k-1} to denote 0.

In order to evaluate the expression given in Eq. (1), the verifier needs to evaluate the multilinear $\hat{g}$ at 2^k points. Computing multilinear evaluations of general functions inherently requires time linear in the domain size (in this case 2^{n+3m}, which is *extremely* large). Thankfully, however, the function g is highly structured. Specifically, we show that g can be evaluated by a very simple model of computation — a *width* 4 *read-once branching program*. Such a branching program may be thought of as an extremely small space streaming algorithm – one that only has 2 bits of storage. The algorithm scans the $(n + 3m)$-bit input x to g in a streaming manner (i.e., bit-by-bit) and, using only 2 bits of storage throughout the entire process, is able to compute $g(x)$.

By a result of Holmgren and Rothblum [HR18], the fact that g is computable by such a branching program suffices to argue that its multilinear extension can be evaluated in time $O(m)$. Combined with Eq. (1), this leads to the $O(2^k \cdot m)$ size verifier of Theorem 1.4.

This is fine as long as the number of columns 2^k is quite small, and already gives the benefits of having a verifier that does not depend on the real heights of the individual columns. Alas, when the number of columns is even mildly larger, this expression can quickly become quite large, especially since the hidden constant in the big O notation depends quadratically on the width of the branching program.

The Jagged Assist: Delegating MLE Computations. A well known fact in the proof-system literature, is that a prover can assist a verifier in reducing a large number $K = 2^k$ of claimed evaluations of a multilinear extension $\hat{f}$ of a function f, to a single evaluation of $\hat{f}$ [GKR15, RR24, CBBZ23]. Using this approach, rather than having the verifier evaluate $\hat{g}$ on the 2^k inputs, we can delegate this computation to the prover and reduce the cost to a single evaluation of $\hat{g}$.

A difficulty that we encounter is that these approaches were designed for the case that f is arbitrary, and so require the prover to run in time $\Omega(2^k \cdot 2^{n+3m})$, which is exorbitant. This runtime seems inherent when f is unstructured, but in our case we consider a highly structured function g for which a single evaluation is exponentially cheaper to compute. Thus, we would like a way to batch verify MLE evaluations such that the prover's runtime scales with the cost of k evaluations of $\hat{g}$.

Following a technique from [RR24, CBBZ23], we show that for general functions $f : \{0,1\}^m \to \mathbb{F}$, it is possible to prove K evaluations of $\hat{f}$ such that the prover only needs to evaluate $\hat{f}$ on $O(m \cdot K)$ different inputs. When $\hat{f}$ is unstructured this leads to $O(m \cdot K \cdot 2^m)$ runtime, which is worse than [CBBZ23]. However, when a single evaluation of the function costs only $O(m)$, we obtain $O(m^2 \cdot K)$ runtime, which is only off by a factor of $O(m)$ from the cost of merely evaluating the function at the K points.

Using this approach, we can already drastically reduce the constant in the verifier's computation in Theorem 1.4, and while increasing the prover's runtime by only $O(m^2 \cdot 2^k)$, which in a typical parameter setting is a negligible amount of additional work.

As a matter of fact, we show that we can do even better by leveraging (yet again) the fact that g is computable by a read-once branching program. For such functions, we show that a natural algorithm used to merely compute the K original evaluations of the branching program, can be used to also prove their correctness with nearly zero overhead. We find this fact to be of independent interest as many simple and structured functions that arise in the construction of efficient proof-systems are computable by such branching programs.

Theorem 1.5. *Let $\hat{f} : \mathbb{F}^m \to \mathbb{F}$ be a multilinear polynomial. Then, there exists an m-round public-coin interactive proof for the language*

$$\left\{ \big((x_1, \ldots, x_K), (y_1, \ldots, y_K)\big) \in (\mathbb{F}^m)^K \times \mathbb{F}^K \ : \ \forall i \in [K],\, \hat{f}(x_i) = y_i \right\},$$

in which the prover and verifier only have oracle access to $\hat{f}$, with soundness error $\frac{2m}{|\mathbb{F}|}$. The prover needs to evaluate f on $O(m \cdot K)$ points and additionally does $O(m \cdot K)$ arithmetic operations. The verifier needs to do only a single evaluation of $\hat{f}$ and in addition does $O(m \cdot K)$ arithmetic operations.

Furthermore, in case f is computable by a width w read-once branching program, given explicit access to this program, the prover can be implemented using $O(m \cdot w^2 \cdot (K + w))$ arithmetic operations.

1.1.2 Handling a Very Large Number of Columns.

Next, we consider the case in which the number of columns is extremely large, and in particular we would like for the verifier to run in time $\ll 2^k$.

Fancy Jagged. We first describe a variant of the basic jagged protocol, which we dub "fancy jagged", which focuses on the case where the columns are not of arbitrary height, but rather correspond to sequences of multiple columns of the same heights — namely tables.

We show a simple extension of the jagged protocol for this setting, in which the verifier work is merely proportional to the number of tables, rather than the number of columns. This extension requires that the table widths are a power of 2.

The fancy jagged protocol has two mild caveats relative to the basic one. First, the underlying branching program in this case has a slightly larger width – namely width 6 rather than 4. This introduces a slight overhead as the branching program evaluation has cost that scales quadratically with the width. Thus, the cost of evaluating it is $(6/4)^2 = 2.25$ larger.

A second caveat is that our solution assumes the the table's width is a power of two. Padding tables that do not have such width can significantly increase the cost. Instead, we propose to deal with larger tables by decomposing them into a sequence of underlying physical tables whose width is a power of two. In this case the verifier cost becomes $\sum_{T \in Tables} \log_2(\#col(T))$.

Committing to the Column Heights. An alternative solution, which we suggest in case the number of columns is very large is for the prover to actually commit to the (cumulative) column heights rather than sending them in the clear. As a matter of fact, this commitment can be done by simply prepending the height data to the dense representation of the polynomial.

We next observe that the computation that the verifier has to do with the column height data is a very low-depth and uniform computation, and so, rather than doing the computation by itself, the verifier can delegate this computation to the prover via the GKR [GKR15] protocol. This approach reduces the verifier's cost to $\text{poly}(k, m)$, while only increasing the prover cost by an additional $O(2^k \cdot m)$ multiplications. While asymptotically very appealing, this solution does introduce some fairly significant complexity to the overall protocol.

1.2 Related Works

The idea of packing multilinears was studied previously (see, e.g., [RRR21, Proposition 3.8] and [Irr24]), but focusing on the case that the column heights are fixed in advance (and in particular the verifier has some branching logic that depends on them).

As already discussed above, the standard method for committing to the computation trace in modern hash-based zkVMs is to commit separately to each column. This introduces significant overhead in the verification process, even when employing optimizations such as batch FRI and placings rows in a common Merkle leaf. We remark that for a moderate number of columns (say in the 1000's), the jagged PCS is (significantly) better than the optimized batch FRI approach, only when the jagged assist protocol of Section 1.1.1 is employed.

Most relevant to our work are constructions of general purpose sparse PCSs from the literature. Recall that in a general purpose sparse PCS the goal is to commit to an arbitrary multilinear $p : \mathbb{F}^n \to \mathbb{F}$ which is non-zero on at most $M \ll 2^n$ locations. In contrast, in this work we construct a sparse PCS only for a particular sparsity structure – jagged functions. Before discussing the

actual constructions, we remark that special purpose sparse PCSs have an automatic benefit over their general-purpose counterparts: in a general sparse PCS it seems necessary to commit to a list of the M non-zero locations. In contrast, in the special purpose setting, the sparsity pattern has a more succinct representation, meaning that, beyond any other benefits, we need to commit to nearly half as much data. In a hash-based setting, this is already a very significant factor.

Spark [Set20] and Surge [STW24] are elegant general-purpose sparse PCSs based on offline memory checking. In Spark and Surge the prover needs to commit to at least 7 polynomials of size M (and 2 polynomials of smaller size).[5] In contrast, in the jagged PCS, the prover does not commit to any additional polynomials.

Spark++ [ST25] is another very recent general-purpose sparse PCS. Whereas Spark and Surge reduce the sparse PCS construction to a dense one, Spark++ assumes that the underlying dense PCSs *commitment* cost is proportional to the sparsity of the data. Spark++ leverages the fact that this is indeed the case in Pedersen based commitments, but also considers extensions to hash based schemes. Based on [ST25, Section 6.4], the cost for Spark++ is dominated by at least $(d^2 + 3d + 2) \cdot M$ multiplications, where d is a parameter that impacts the verification time.[6]. It seems that in our case, when the data size is very large (e.g. the entire computation trace), one would need at the very least $d \geq 2$, which translates into $12M$ multiplications.

SpeedySpartan [ST25], similarly to the jagged PCS, is a sparse PCS for a particular form of sparse polynomials — specifically, where there is exactly one 1 per row. V-SPARK [SR25] is also an extension of Spark for particular sparse polynomials but from a very large class – specifically composed of recursively embedded blocks. Their approach can also yield a jagged PCS, although we suspect that the greater level of generality might bear some additional cost.

2 Preliminaries

2.1 Multilinear Extension

Let $\mathbb{F}$ be a finite field and $m \in \mathbb{N}$ be an integer. For every function $f : \{0,1\}^m \to \mathbb{F}$ there exists a unique multilinear polynomial $\hat{f} : \mathbb{F}^m \to \mathbb{F}$ that agrees with f on $\{0,1\}^m$. We refer to $\hat{f}$ as the *multilinear extension* of f.

The multilinear extension $\hat{f}$ can be represented explicitly as:

$$\hat{f}(z) = \sum_{b \in \{0,1\}^m} eq(z,b) \cdot f(b), \tag{2}$$

for every $z \in \mathbb{F}^m$, where $eq(z,b) = \prod_{i \in [m]} eq_1(z_i, b_i)$ and $eq_1(z_i, b_i) = z_i \cdot b_i + (1 - z_i) \cdot (1 - b_i)$.

We will extensively use the following basic fact (which follows from the uniqueness of the multilinear extension):

Fact 2.1. *Two multilinear polynomials $f, g : \mathbb{F}^m \to \mathbb{F}$ that agree on every input in $\{0,1\}^m$ must also agree on every input in $\mathbb{F}^m$.*

We will also use an efficient algorithm due to Vu *et al.* [VSBW13], as optimized in [DT24, Algorithm 1] (see also concrete improvements in [CFFZ24, Rot24]), for generating the sequence of eq values.

[5] In case the prover is allowed to run in time proportional to 2^n then it is possible to use Surge with fewer commitments. However, this does not seem useful in the in the typical scenario where $2^n \gg M$.

[6] More specifically, this is the cost for the Shout PIOP that is used within Spark++

Proposition 2.2 ([VSBW13, DT24]). *Given $z \in \mathbb{F}^m$, the sequence of values $(eq(z,b))_{b\in\{0,1\}^m}$ can be generated using 2^m multiplications and 2^m additions.*

2.2 The Sumcheck Procotol

We use the celebrated sumcheck protocol [LFKN92], specialized to the setting of a product of two multilinear polynomials, while precisely accounting for the prover cost.

Lemma 2.3. *There exists an m-round interactive protocol between a prover and verifier, in which the prover gets as input functions $f, g : \{0,1\}^m \to \mathbb{F}$ and the verifier gets as input a value v. At the end of the protocol the verifier either rejects the proof, or outputs $z' \in \mathbb{F}^m$ and $v' \in \mathbb{F}$ such that:*

1. ***Completeness:*** *if $v = \sum_{x\in\{0,1\}^m} f(x) \cdot g(x)$ then the verifier outputs $(z', v') \in \mathbb{F}^m \times \mathbb{F}$ such that $v' = \hat{f}(z') \cdot \hat{g}(z')$.*
2. ***Soundness:*** *if $v \neq \sum_{x\in\{0,1\}^m} f(x) \cdot g(x)$ then when interacting with any possibly cheating prover, with probability $1 - \frac{2m}{|\mathbb{F}|}$, the verifier either rejects or outputs $(z', v') \in \mathbb{F}^m \times \mathbb{F}$ such that $v' \neq \hat{f}(z') \cdot \hat{g}(z')$.*
3. ***Efficiency:*** *The verifier performs $O(m)$ field operations. The prover does $4 \cdot 2^m$ multiplications and $O(2^m)$ basic operations.*

Setty and Thaler [ST25, Section 3.3] precisely account for the prover cost in the sumcheck protocol when used in the context of a zerocheck procedure. For completeness, we analyze the "vanilla" sumcheck of Lemma 2.3 in Appendix A (but emphasize that we use a standard sumcheck protocol, with the usual optimizations).

3 Jagged Polynomial Commitment

In this section we give our main construction of an efficient MLPCS for (multilinear extensions of) jagged functions. Or, more precisely, an efficient reduction from a jagged PCS to a standard one, thereby proving Theorem 1.4. Throughout this section we identify binary strings with integers in the natural way.

We consider a jagged polynomial $p : \mathbb{F}^n \times \mathbb{F}^k \to \mathbb{F}$ with heights $\big(h_y \in [0, \ldots, 2^n - 1]\big)_{y\in\{0,1\}^k}$. Let $t = (t_y)_{y\in\{0,1\}^k}$ be the corresponding cumulative height sequence and let $\hat{q} : \mathbb{F}^m \to \mathbb{F}$ be the corresponding dense representation of $\hat{p}$. Recall that $\hat{p} = sparse(\hat{q}, t)$. Given t, there is a bijection $i \mapsto (row_t(i), col_t(i)$ between $\{0,1\}^m$ and the non-zero part of p via two functions $row_t : \{0,1\}^m \to \{0,1\}^n$ and $col_t : \{0,1\}^m \to \{0,1\}^k$.

Our goal is to reduce an evaluation claim of the form $\hat{p}(z_r, z_c) = v$, where $z_r \in \mathbb{F}^n$, $z_c \in \mathbb{F}^k$ and $v \in \mathbb{F}$ to an evaluation claim on $\hat{q}$. To do so, observe that:

$$\hat{p}(z_r, z_c) = \sum_{\substack{x\in\{0,1\}^n \\ y\in\{0,1\}^k}} p(x,y) \cdot eq(x, z_r) \cdot eq(y, z_c) = \sum_{i\in\{0,1\}^m} q(i) \cdot eq(row_t(i), z_r) \cdot eq(col_t(i), z_c), \quad (3)$$

where the first equality is by the definition of the multilinear extension, and the second by the fact that p is a jagged function, the definition of q, and Fact 1.3.

Define a function $f_t : \{0,1\}^n \times \{0,1\}^k \times \{0,1\}^m \to \{0,1\}$ such that $f_t(z_r, z_c, i) = 1$ if and only if $row_t(i) = z_r$ and $col_t(i) = z_c$. We next show that for every index $i \in \{0,1\}^m$ and $z_r \in \mathbb{F}^n$ and $z_c \in \mathbb{F}^k$, it holds that

$$\hat{f}_t(z_r, z_c, i) = eq(row_t(i), z_r) \cdot eq(col_t(i), z_c), \tag{4}$$

where $\hat{f}_t$ is the multilinear extension of f_t. This follows from Fact 2.1, since for every fixed index i, both sides of the equation are multilinear in z_c and z_r, and equality clearly holds (by definition of f and eq) when both are Boolean valued.

Remark 3.1. *We emphasize that while Eq. (4) holds for $i \in \{0,1\}^m$, it is unlikely to hold for general $i \in \mathbb{F}^m$ as the RHS has higher degree in i.*

Thus, using Eqs. (3) and (4), our claim now simplifies to proving that:

$$v = \sum_{i \in \{0,1\}^m} \hat{q}(i) \cdot \hat{f}_t(z_r, z_c, i).$$

This is done by running the sumcheck protocol for products of multilinears (i.e., Lemma 2.3). The sumcheck protocol reduces the above to a claim about a single point of $\hat{q}$ and a single point of $\hat{f}_t$. The former claim is the output of the verifier, and we show how the verifier can compute the latter claim directly.

Completeness and soundness follow immediately from Lemma 2.3, and so we proceed to the complexity analysis.

On the prover's side, recall that the prover already has access to $\big(q(i)\big)_{i \in \{0,1\}^m}$. It can generate the sequence of values $\big(\hat{f}_t(z_r, z_c, i)\big)_{i \in \{0,1\}^m}$ using Eq. (4) by first using the algorithm of Proposition 2.2 to generate the sequences $eq(z_r, \cdot)$ and $eq(z_c, \cdot)$ and, given these, the sequence above can be generated using 2^m additional multiplications (by simply iterating over $i \in \{0,1\}^m$). The total work to generate f is therefore $2^m + 2^n + 2^k$ multiplications and $O(2^n + 2^k)$ additions. By Lemma 2.3, the sumcheck itself requires an additional $4 \cdot 2^m$ multiplications and $O(2^m)$ basic operations.

During the jagged sumcheck, the amount of arithmetic work for the verifier is $O(m)$. At the end of the sumcheck, the verifier derives one claim of the form $\hat{q}(i) = \alpha$ and one of the form $\hat{f}(i) = \beta$, for $i \in \mathbb{F}^m$ and $\alpha, \beta \in \mathbb{F}$. The former is the claim about $\hat{q}$ that is the output of the verifier in the jagged protocol. For the latter (i.e., the claim about $\hat{f}$), we show in the following section how it can be computed efficiently by the verifier.

Proposition 3.2. *Given cumulative heights t and a point $z \in \mathbb{F}^{n+k+m}$, the function $\hat{f}_t(z)$ can be computed using $O(m \cdot 2^k)$ arithmetic operations.*

Observe that given Proposition 3.2 we get that the verifier's work in the sumcheck overall is $O((m+n) \cdot 2^k)$ and Theorem 1.4 follows.

3.1 Proof of Proposition 3.2: Efficiently Computing $\hat{f}_t$

Let $g : \{0,1\}^n \times \{0,1\}^{3m} \to \{0,1\}$ be defined as $g(a,b,c,d) = 1$ if and only if $b < d$ and $b = a + c$. Let $\hat{g} : \mathbb{F}^{n+3m} \to \mathbb{F}$ denote the multilinear extension of g. We relate $\hat{f}_t$ to $\hat{g}$ via the following claim.

Claim 3.2.1. *For every $z_r \in \mathbb{F}^n$, $z_c \in \mathbb{F}^k$ and $i \in \mathbb{F}^m$ it holds that:*

$$\hat{f}_t(z_r, z_c, i) = \sum_{y \in \{0,1\}^k} eq(z_c, y) \cdot \hat{g}(z_r, i, t_{y-1}, t_y), \tag{5}$$

where, to simplify the notation, we use t_{0^k-1} to denote 0.

Proof. Observe that both sides of the equation are multilinear in z_r, z_c, and i. Thus, by Fact 2.1, it suffices to prove the claim when $z_r \in \{0,1\}^n$, $z_c \in \{0,1\}^k$ and $i \in \{0,1\}^m$. In this case, the RHS simplifies to $g(z_r, i, t_{y-1}, t_y)$, where $y = z_c$.

Suppose $g(z_r, i, t_{y-1}, t_y) = 1$. Then, by definition, $i < t_y$ and $i = z_r + t_{y-1}$ (where t_{0^k-1} is defined as 0). As t is admissible, this means that $t_{y-1} \leq i < t_y$ and that $z_r = i - t_{y-1}$. Thus, $col_t(i) = y$ and $row_t(i) = z_r$ and so $f_t(z_r, y, i) = 1$.

Similarly, if $f_t(z_r, y, i) = 1$ then $col_t(i) = y$ and $row_t(i) = z_r$, which means that $t_{y-1} \leq i < t_y$ and that $z_r = i - t_{y-1}$, and therefore $g(z_r, i, t_{y-1}, t_y) = 1$. □

Thus, by Claim 3.2.1, to compute $\hat{f}$ it suffices to compute $\hat{g}$ at 2^k points. Computing the multilinear extension of a general function on $n + 3m$ variables takes time 2^{n+3m}, which we cannot afford. Rather, in order to compute $\hat{g}$ efficiently, we next show that there exists a small-width read-once branching program for computing g. This, in combination with Lemma 4.2 below, implies that the multilinear extension $\hat{g}$ can be efficiently computed.

Claim 3.2.2. *The function g is computable by a width-4 read-once branching program.*

Proof. Recall that $n \leq m$. Wlog we assume $n = m$ (otherwise we can just pad the a part of the input with zeros). We will use the fact that the branching program is allowed to read the input in any fixed order. Specifically, we rearrange the input to g to consist of symbols over the alphabet $\Sigma = \{0,1\}^4$ consisting of the 4 constituent bits a, b, c, d. Thus, $g : (\{0,1\}^4)^m \to \{0,1\}$ is defined as follows: given input $(a_i, b_i, c_i, d_i)_{i \in \{0,\ldots,m-1\}}$ output 1 if and only if $b < d$ and $b = a + c$, where $a = \sum_i a_i 2^i$ and similarly for b, c and d.

For every bit i that is read from the four parts of the input, the branching program keeps track of two registers (both have an initial value of 0):

- The next carry bit arising from the integer addition of the i-bit prefix of a with the corresponding prefix of c. Denote this bit by $carry_i$ (where $carry_0 = 0$).
- Is the i-bit prefix of b smaller than the i-bit prefix of d (as integers)? Denote this bit by lt_i (and set the initial $lt_0 = 0$).

Given the registers $carry_i$ and lt_i, and the input bits $a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1}$, we show how to update the registers. First, in case we find an inconsistency in the addition (namely, if the LSB of $a_{i+1} + c_{i+1} + carry_i$ is not equal to b_{i+1}) then the branching program can immediately move to a rejecting sink. In case the addition was consistent, we can update $carry_{i+1}$ to be the MSB of $a_{i+1} + c_{i+1} + carry_i$. The branching program can update the second register lt_{i+1} to 1 if and only if either $(d_i = 1) \wedge (b_i = 0)$ or if $(d_i = b_i) \wedge lt_i = 1$.

As our registers can take on a total of $2^2 = 4$ possible values, the width of the branching program is 4. □

Proposition 3.2 now follows directly by combining Claim 3.2.2 and Lemma 4.2.

4 Multilinear Extensions of Read-Once Branching Programs

Holmgren and Rothblum [HR18] showed an efficient procedure for computing the multilinear extension of functions that are computable by small-width read-once branching programs. Their focus is on matrix branching programs. In this section we give a self contained overview of their result when applied to standard read-once branching programs and then also present their result for matrix branching programs. We extend their matrix branching program result to handle a "symbolic evaluation" in which some parts of the program are left as indeterminates (this will be useful later on for batching program of their MLE evaluations).

A *read-once branching program* (ROBP) is a computational model that captures bounded space computations, in which the input is read in a one-pass streaming manner. While this model is weak, it is very convenient for capturing simple calculations that often arise in proof systems. For example, checking whether two strings are equal, the first is greater than the second (when interpreted as integers), addition relations, etc.

Formally, the model is defined as follows:

Definition 4.1. *For $w \in \mathbb{N}$, a* width-w read-once branching program *(ROBP) over an alphabet $\Sigma = \{0,1\}^b$ consists of a layered directed graph $G = (V, E)$, with $n+1$ layers. The first layer, denoted layer 0, consists of a single vertex known as the* source. *Each subsequent layer consists of at most w vertices. The vertices in the last layer (aka layer n) are called* sinks *and are each labeled by a field element.*

Each vertex v in layer $i < n$ has exactly 2^b outgoing edges, labeled by all of the strings in Σ, each of which goes into layer $i+1$ (or directly to a sink). For every symbol $\sigma \in \{0,1\}^b$, we denote by $\Gamma(v, \sigma)$ the neighbor of v that is incident to the outgoing edge that is labeled by σ.

Given an input $x \in \Sigma^n$, the ROBP is evaluated by starting at the source and proceeding on the edge marked with x_1, then with x_2, and so forth. After n such steps, we reach a sink and output its label. Intuitively, such a branching program captures a streaming algorithm with space $\log_2(w)$, where the vertices in layer i represent all possible configurations of the memory at time step i.

The following lemma shows how to efficiently compute the MLE of functions computable by this model.

Lemma 4.2 ([HR18])**.** *Suppose that the function $f : (\{0,1\}^b)^n \to \mathbb{F}$ can be computed by a width w ROBP. Then $\hat{f} : (\mathbb{F}^b)^n \to \mathbb{F}$ can be computed using $n \cdot (w^2 + 2^b)$ multiplications and $O(n \cdot w \cdot 2^b)$ additions.*

Since, as stated, the lemma is not explicitly proved in [HR18], we provide the full proof.

Proof. For a vertex v in layer i, denote by $f_v : (\{0,1\}^b)^{n-i} \to \mathbb{F}$ the function obtained by running the ROBP, but starting at v (rather than the original source vertex). Our algorithm, given as input a vector $z \in (\mathbb{F}^b)^n$, processes the branching program in reverse order: namely, from the sinks back to the source. The algorithm computes $\hat{f}_v$ applied to the relevant suffix of z, by using values that it already previously computed for the vertices in layer $i+1$. Eventually it obtains $\hat{f}_{src}(z)$, where *src* is the source vertex, which is precisely equal to the desired output $\hat{f}(z)$.

We proceed to describe how the algorithm inductively computes $\hat{f}_v$ applied to the relevant suffix of z (by reverse induction). By definition, if v is a sink labeled by the field element α, then it holds that $\hat{f}_v = \alpha$. For non-sink vertices v we can compute $\hat{f}_v$ using the following claim:

Claim 4.2.1. *For every vertex v in layer $i < n$, symbol $\zeta \in \mathbb{F}^b$, and vector $z \in (\mathbb{F}^b)^{n-i-1}$ it holds that:*

$$\hat{f}_v(\zeta, z) = \sum_{\sigma \in \{0,1\}^b} eq(\zeta, \sigma) \cdot \hat{f}_{\Gamma(v,\sigma)}(z).$$

Proof. Observe that both sides of the equation are multilinear (in the variables ζ and z). For Boolean-valued $\zeta \in \{0,1\}^b$ and $z \in (\{0,1\}^b)^n$ the equation clearly holds, since in this case:

$$\sum_{\sigma \in \{0,1\}^b} eq(\zeta, \sigma) \cdot \hat{f}_{\Gamma(v,\sigma)}(z) = \hat{f}_{\Gamma(v,\zeta)}(z) = \hat{f}_v(\zeta, z).$$

The claim now follows from Fact 2.1. □

Using Claim 4.2.1, the algorithm proceeds layer by layer, and computes the evaluation of $\hat{f}_v$ for every vertex v (on the corresponding suffix of z). Eventually, the value obtained at the source vertex is precisely the desired $\hat{f}(z)$.

To obtain the desired number of multiplications we observe that for vertex v in layer $i < n$, the equation in Claim 4.2.1 can be rewritten as:

$$\hat{f}_v(\zeta, z) = \sum_{v' \text{ in layer } i+1} \hat{f}_{v'}(z) \cdot \sum_{\sigma \in \{0,1\}^b \text{ s.t. } \Gamma(v,\sigma)=v'} eq(\zeta, \sigma). \tag{6}$$

Thus, the algorithm first computes $eq(\zeta, \cdot)$ using Proposition 2.2. Then, by enumerating all v in layer i and v' in layer $i+1$, and $\sigma \in \{0,1\}^b$, it can compute $\left(\sum_{\sigma \in \{0,1\}^b \text{ s.t. } \Gamma(v,\sigma)=v'} eq(\zeta, \sigma)\right)_{v'}$. Finally for every v, it uses Eq. (6) to compute $\hat{f}_v(\zeta, z)$ using an additional w multiplications. □

4.1 Matrix Branching Program

We will also consider a variant of the branching program model known as *matrix branching programs* (also considered in [HR18]).

A *read-once matrix branching program* (MBP) is defined by a sequence of $w \times w$ dimensional matrices $M = \left(M_j^{(\sigma)} \in \mathbb{F}^{w \times w}\right)_{\sigma \in \{0,1\}^b, j \in [n]}$ together with a *sink vector* $u \in \mathbb{F}^w$. Given an input $x \in (\{0,1\}^b)^n$, the output of the MBP is the first entry of the vector $\left(\prod_{j=1}^n M_j^{(x_j)}\right) \cdot u$. That is, the output is $(e_1)^T \cdot \left(\prod_{j=1}^n M_j^{(x_j)}\right) \cdot u$, where e_1 is the unit vector $(1, 0, \ldots, 0)$. In other words, the input bits act as selectors for matrices for every layer, and the MBP is evaluated by taking the product of the selected matrices (in the right order).

We refer to the branching program determined by $M = \left(M_j^{(\sigma)}\right)_{\sigma \in \{0,1\}^b, j \in [n]}$ and u by (M, u) and refer to w as its *width*.

Remark 4.3. *A read-once branching program of Definition 4.1 can be represented as an MBP. Indeed, for every pair of layers i and $i+1$, one can write down, for every alphabet symbol $\sigma \in \Sigma$, a matrix $M_i^{(\sigma)}$ which describes the transition function of the branching program from any unit vector representing the evaluation at layer i to the unit vector describing the evaluation at layer $i+1$.*

Moreover, in this case the matrices $M_i^{(\sigma)}$ are multiplication friendly *in the sense that multiplication of any matrix $X \in \mathbb{F}^{w \times w}$ with a multiplication friendly matrix M can be done using just a linear number of additions, and no multiplications.*

We next show how to evaluate the multilinear extension of this generalization.

Lemma 4.4 (MBP Multilinear Evaluation)**.** *Suppose that $f : (\{0,1\}^b)^n \to \mathbb{F}$ is computable by a width w matrix branching program. Then $\hat{f} : (\mathbb{F}^b)^n \to \mathbb{F}$ is computable using $O(n \cdot w^2 \cdot (2^b + w))$ arithmetic operations.*

Lemma 4.4 follows immediately from the following lemma, which shows how to evaluate the MBP even when the sink vector is not provided. We refer to such an evaluation as a symbolic evaluation. This symbolic evaluation will be useful for us later on, when batch proving multiple evaluations of the MBP.

Lemma 4.5 (MBP Symbolic Evaluation)**.** *There exists an algorithm that given only the matrices $(M_j^{(\sigma)})_{\sigma\in\{0,1\}^b, j\in[n]}$ and input $z \in (\mathbb{F}^b)^n$, but independently of the sink vector u, outputs a vector $res \in \mathbb{F}^w$ so that for a given sink vector $u \in \mathbb{F}^w$ it holds that $\hat{f}_{M,u}(z) = \langle res, u\rangle$, where $\hat{f}_{M,u}$ is the multilinear extension of the branching program determined by (M, u). The algorithm uses $O(n \cdot w^2 \cdot (2^b + w))$ arithmetic operations. In case the matrices are multiplication friendly, this reduces to $O(n \cdot w^2 \cdot 2^b)$ arithmetic operations.*

Proof. Let $M = \left(M_j^{(\sigma)}\right)_{\sigma\in\{0,1\}^b, j\in[n]}$. For an input $z \in (\mathbb{F}^b)^n$, the algorithm computes, by reverse recursion (i.e., from n to 0), the matrices $A_0, \ldots, A_n \in F^{w\times w}$, where $A_n = I_w$ is the identity matrix. For $j \in [n]$ it holds that

$$A_{j-1} = \left(\sum_{\sigma\in\{0,1\}^b} eq(z_j, \sigma) \cdot M_j^{(\sigma)}\right) \cdot A_j.$$

Observe that, using the above equation, A_{j-1} can be computed from A_j using $O(w^2 \cdot (2^b + w))$ arithmetic operations (the sequence of values $eq(z_j, \cdot)$ is computed using Proposition 2.2, and we use the naive $O(w^3)$-time matrix multiplication algorithm). If the matrices are multiplication friendly the cost reduces to $.O(w^2 \cdot 2^b)$.

We next show that res, defined as $(e_1)^T \cdot A_0$, satisfies the requirements of the lemma. For a given sink vector $u \in \mathbb{F}^w$, observe that:

$$\langle res, u\rangle = (e_1)^T \cdot A_0 \cdot u = (e_1)^T \cdot \prod_{j=1}^{n} \left(\sum_{\sigma\in\{0,1\}^b} eq(z_j, \sigma) \cdot M_j^{(\sigma)}\right) \cdot u \tag{7}$$

Next, we show that the RHS of Eq. (7) is equal to $\hat{f}_{M,u}(z)$, where $\hat{f}_{M,u}$ is the multilinear extension of the MBP determined by (M, u).

To see that the above is indeed the case, observe that both the RHS of Eq. (7) and $\hat{f}_{M,u}(z)$ are multilinear in z, and that for Boolean valued inputs $z \in (\{0,1\}^b)^n$ they are equal, since in this case:

$$\prod_{j=1}^{n} \left(\sum_{\sigma\in\{0,1\}^b} eq(z_j, \sigma) \cdot M_j^{(\sigma)}\right) = \prod_{j=1}^{n} M_j^{(z_j)}$$

and so, for such inputs, by Eq. (7):

$$\langle res, u\rangle = (e_1)^T \cdot \left(\prod_{j=1}^{n} M_j^{(z_j)}\right) \cdot u = \hat{f}_{M,u}(z).$$

The claim now follows from Fact 2.1. □

We use the symoblic evaluation algorithm to generate a very particular sequence of related MLE evaluations. This will be useful later on in the context of batch proving of such evaluations.

Lemma 4.6. *Let $h : (\{0,1\}^b)^m \to \mathbb{F}$ be computable by a width w read-once matrix branching program, and let Λ be a set of distinct field elements.*

There exists an algorithm that given as input $x_1, \ldots, x_k \in \mathbb{F}^n$ and streaming access to $\rho \in \mathbb{F}^n$ operates as follows. After reading the $(j-1)$-th bit of ρ the algorithm outputs:

$$\left(h(\rho[1,\ldots,j-1], \lambda, x_i[j+1,..,m])\right)_{\lambda \in \Lambda}.$$

The algorithm uses $O(n \cdot w^2 \cdot (k \cdot 2^b + |\Lambda| \cdot w))$ arithmetic operations.

Proof. The algorithm follows a similar strategy to that in Lemma 4.5.

Let $M = \left(M_j^{(\sigma)}\right)_{\sigma \in \{0,1\}^b, j \in [n]}$. For every $i \in [k]$, the algorithm computes, by reverse recursion (i.e., from n to 0), the matrices $A_0^{(i)}, \ldots, A_n^{(i)} \in F^{w \times w}$, where for every $i \in [k]$, it holds that $A_n^{(i)} = I_w$ is the identity matrix and for $j \in [n]$ it holds that

$$A_{j-1}^{(i)} = \left(\sum_{\sigma \in \{0,1\}^b} eq(x_i[j], \sigma) \cdot M_j^{(\sigma)} \right) \cdot A_j^{(i)}.$$

Similarly to Lemma 4.5, and using the fact that the matrices are multiplication friendly (see Remark 4.3), $A_{j-1}^{(i)}$ can be computed from $A_j^{(i)}$ using $O(w^2 \cdot 2^b)$ arithmetic operations.

Now, for each round $j \in [n]$, we are given $\rho[1, \ldots, j-1]$. Assume we also at this point have generated a matrix B_{j-} such that:

$$B_{j-1} = \prod_{j'=1}^{j-2} \left(\sum_{\sigma \in \{0,1\}^b} eq(\rho_{j'}, \sigma) \cdot M_{j'}^{(\sigma)} \right)$$

(we will see how to maintain this invariant). Now for every $\lambda \in \Lambda$, the algorithm outputs as the claimed evaluation of $h(\rho[1, \ldots, j-1, \lambda, x^{(i)}[j+1])$ the value:

$$(e_1)^T B_{j-1} \cdot \left(\sum_{\sigma \in \{0,1\}^b} eq(\lambda, \sigma) \cdot M_{j'}^{(\sigma)} \right) \cdot A_{n-j}^{(i)} \cdot u.$$

In order to generate the next B_j, given ρ_j, the algorithm also computes:

$$B_j = B_{j-1} \cdot \left(\sum_{\sigma \in \{0,1\}^b} eq(\rho_j, \sigma) \cdot M_{j'}^{(\sigma)} \right)$$

It follows immediately from the definition of B_j that the invariant is maintained. We proceed to show that $h(\rho[1, \ldots, j-1, \lambda, x^{(i)}[j+1])$ was computed correctly. Since h is multilinear, it suffices, by Fact 2.1, to show this in the case that all of the inputs (i.e., λ, ρ and the x_i's) are Boolean-valued. In this case:

$$B_{j-1} \cdot \left(\sum_{\sigma \in \{0,1\}^b} eq(\lambda, \sigma) \cdot M_{j'}^{(\sigma)} \right) \cdot A_{n-j}^{(i)} = B_{j-1} \cdot M_{j'}^{(\lambda)} \cdot A_{n-j}^{(i)}$$

$$= \prod_{j'=1}^{j-2} \left(\sum_{\sigma \in \{0,1\}^b} eq(\rho_{j'}, \sigma) \cdot M_{j'}^{(\sigma)} \right) \cdot M_{j'}^{(\lambda)} \cdot A_{n-j}^{(i)}$$

$$= \left(\prod_{j'=1}^{j-2} M_{j'}^{(\rho_{j'})} \right) \cdot M_{j'}^{(\lambda)} \cdot A_{n-j}^{(i)}$$

$$= \left(\prod_{j'=1}^{j-2} M_{j'}^{(\rho_{j'})} \right) \cdot M_{j'}^{(\lambda)} \cdot \prod_{j'=j+1}^{n} \left(\sum_{\sigma \in \{0,1\}^b} eq(x_i[j'], \sigma) \cdot M_{j'}^{(\sigma)} \right)$$

$$= \left(\prod_{j'=1}^{j-2} M_{j'}^{(\rho_{j'})} \right) \cdot M_{j'}^{(\lambda)} \cdot \left(\prod_{j'=j+1}^{n} M_{j}^{(x_i[j'])} \right)$$

and the lemma now follows by observing that:

$$(e_1)^T \cdot \left(\prod_{j'=1}^{j-2} M_{j'}^{(\rho_{j'})} \right) \cdot M_{j'}^{(\lambda)} \cdot \left(\prod_{j'=j+1}^{n} M_{j}^{(x_i[j'])} \right) \cdot u = h(\rho[1, \ldots, j-1], \lambda, x_i[j+1, \ldots, n])$$

The cost of the algorithm is generating the $A_j^{(i)}$ matrices, which can be done in a total of $O(k \cdot n \cdot 2^b \cdot w^2)$ arithmetic operations. Generating each B_j matrix similarly requires $(2^b \cdot w^2)$ work, for every $j \in [n]$. The final iteration to generate the output requires $O(w^3)$ arithmetic, per layer, due to the matrix multiplication.[7]

The total time number of operations is therefore $O(k \cdot n \cdot 2^b \cdot w^2 + n \cdot w^3) = O(n \cdot w^2 \cdot (k \cdot 2^b + w))$. □

5 Batch Proving of Multilinear Evaluations

In this section we show how a prover can prove to a verifier the correctness of some k evaluations of a multilinear polynomial $h : \mathbb{F}^m \to \mathbb{F}$, so that the verifier only needs to evaluate the polynomial at one point.

The goal here is closely related to the goal in the 2-to-1 trick of [GKR15], as well as alternate procedures such as those described in [RR24, CBBZ23]. The key distinction is that, in these works, the prover was allowed to run in time $\Omega(2^m)$, which is necessary when h is arbitrary. In contrast, here, rather we will count the number of evaluations of h that the prover needs to perform, which will be extremely beneficial when h is a simple function that can be computed in time $o(2^m)$ (e.g., if h is computable by a small-width branching program).

Lemma 5.1. *Let $h : \mathbb{F}^m \to \mathbb{F}$ be a multilinear polynomial. There exists an m-round public-coin interactive proof for the language*

$$\Big\{ \big((x_1, \ldots, x_k), (y_1, \ldots, y_k) \big) \in (\mathbb{F}^m)^k \times \mathbb{F}^k \ : \ \forall i \in [k],\, h(x_i) = y_i \Big\},$$

[7] Given that we plan to only use very small values of w, we avoid using advanced matrix multiplication algorithms.

with soundness error $\frac{2m}{|\mathbb{F}|}$. The prover needs to evaluate h on $O(m \cdot k)$ points and additionally does $O(m \cdot k)$ arithmetic operations. The verifier needs to do only a single evaluation of h and in addition does $O(m \cdot k)$ arithmetic operations.

Furthermore, in round $j \in [m]$ the prover only needs to evaluate h on the points $\{(\rho, \lambda, x_i[j+1, .., m])\}_{\lambda \in \Lambda})$, where $\rho \in \mathbb{F}^{j-1}$ is the randomness sent by the verifier in the previous rounds and Λ is a set of three distinct field elements.

Lemma 5.1 implies Theorem 1.5, except for the furthermore part, which we explain immediately after the proof of the lemma.

Proof. Let $(x_1, \ldots, x_k), (y_1, \ldots, y_k)) \in (\mathbb{F}^m)^k \times \mathbb{F}^k$ be an input. Recall that the goal is to check that $h(x_i) = y_i$, for every $i \in [k]$. As its first step, the verifier chooses at random[8] $r_1, \ldots, r_k \in \mathbb{F}$ and the claim is reduced to checking that:

$$\sum_{i \in [k]} r_i \cdot h(x_i) = \sum_{i \in [k]} r_i \cdot y_i.$$

Note that in case the original claim was false, with probability $1 - 1/|\mathbb{F}|$, the new claim is also false.

The RHS of the above equation can be computed directly by the verifier using $O(k)$ field operations. For the LHS, using the definition of the multilinear equation, we can rewrite it as:

$$\sum_{i \in [k]} r_i \sum_{b \in \{0,1\}^m} eq(b, x_i) \cdot h(b) = \sum_{b \in \{0,1\}^m} h(b) \sum_{i \in [k]} r_i \cdot eq(b, x_i).$$

Thus, the prover and verifier engage in the sumcheck protocol of Lemma 2.3 to compute $\sum_{b \in \{0,1\}^m} P(b)$, where $P(b) = h(b) \cdot \sum_{i \in [k]} r_i \cdot eq(b, x_i)$.

Completeness and soundness follow immediately from Lemma 2.3, and so we proceed to an accounting of the prover and verifier costs.

For the verifier, the sumcheck protocol itself only requires $O(m)$ arithmetic operations. At the end of the sumcheck, the verifier needs to compute $P(\rho) = h(\rho) \cdot \sum_{i \in [k]} r_i \cdot eq(\rho, x_i)$, where $\rho \in \mathbb{F}^m$ is the verifier's randomness in the sumcheck protocol. Thus, the verifier only needs to perform a single evaluation of h plus an additional $O(k \cdot m)$ arithmetic work to compute $\sum_{i \in [k]} r_i \cdot eq(\rho, x_i)$.[9]

We next turn to the prover cost. Let Λ be a set of three distinct field elements to be used as the evaluation points in the sumcheck protocol.[10] In the j-th round of sumcheck, for $j \in [m]$, a sequence $\rho \in \mathbb{F}^{j-1}$ will have already been fixed by the verifier challenges from the previous rounds.

[8] One could instead employ a small-bias generator. For example, by choosing a single $r \in \mathbb{F}$ and setting $r_i = r^i$. This introduces an additional soundness error of $k/|\mathbb{F}|$.

[9] This computation can be optimized by breaking the m long eq evaluation, which is repeated k times, into $m/\log(k)$ chunks, each of size $\log(k)$ and employing the algorithm of Proposition 2.2 on each chunk. However, this requires the verifier to work as a random access machine, rather than an arithmetic circuit, and is therefore somewhat incompatible with recursion.

[10] Using an optimization due to Gruen [Gru24], it actually suffices for Λ to be a set of size 2 (as long as it is not specifically the set $\{0, 1\}$)

To compute its j-th message, the prover needs to compute for every $\lambda \in \Lambda$:

$$\begin{aligned}\sum_{b\in\{0,1\}^{m-j}} P(\rho,\lambda,b) &= \sum_{i\in[k]} r_i \sum_{b\in\{0,1\}^{m-j}} h(\rho,\lambda,b)\cdot eq((\rho,\lambda,b),x_i)\\ &= \sum_{i\in[k]} r_i \cdot eq((\rho,\lambda),x_i[1,\ldots,j]) \sum_{b\in\{0,1\}^{m-j}} h(\rho,\lambda,b)\cdot eq(b,x_i[j+1,\ldots,m])\\ &= \sum_{i\in[k]} r_i \cdot eq\big((\rho,\lambda),x_i[1,\ldots,j]\big)\cdot h\big(\rho,\lambda,x_i[j+1,\ldots,m])\big).\end{aligned}$$

Thus, the j-th round message can be computed using k evaluations of h and an additional $O(k)$ arithmetic work, by reusing the partial evaluation of eq from the previous round. The furthermore part of the lemma follows immediately from the construction. □

Using the algorithm of Lemma 4.6 to produce the prover messages within the protocol of Lemma 5.1 yields the furthermore part of Theorem 1.5.

6 Fancy Jagged: When Multiple Columns have the Same Height

In this section we consider a variant of the jagged PCS in which the columns are grouped into "tables" which are simply a sequence of columns of the same height. Due to a technical limitation, we restrict our attention to tables in which the number of columns (also referred to as the "width") is always a power of two. Rather than employing expensive padding, tables with other sizes can and should be handled by decomposing them into a logarithmic sequence of tables whose width is a power of 2. For example, a table of width 9 can be represented by two tables: one of width 8 and one of width 1.

We adapt the notation and definitions from Section 3 as follows. We use 2^k to denote the number of *tables*. Each table $y \in \{0,1\}^k$ consists of some 2^{c_y} columns for an integer c_y, and the height of each table is h_y. We emphasize that c_y denotes the *logarithm* of table y's width.

We use $c = \max_y c_y$ so that the maximal number of columns in each table is 2^c.

The accumulative table heights $t = (t_y)_{y\in\{0,1\}^k}$ are now defined as $t_y = \sum_{y'<y} 2^{c_{y'}} \cdot h_{y'}$. We refer to the sequence $(t_{0^k},\ldots,t_{1^k},c_{0^k},\ldots,c_{1^k})$ as a *configuration* of the sparse polynomial since it determines the "non-zero" part of it.

Given the configuration, the number of non-zeros entries in the sparse polynomial p is now $M = \sum_y 2^{c_y}\cdot h_y$. Similarly to Section 3, we assume (via padding) that M is a power of 2, and denote $m = \log_2(M)$.

For a fixed configuration, we associate indices $i \in \{0,1\}^m$ with triples (tab, row, col), where $tab \in \{0,1\}^k$, $row \in \{0,1\}^n$, and $col \in \{0,1\}^c$, by extending the bijection described in Section 3 in the natural way. We use this mapping to define corresponding functions $tab : \{0,1\}^m \to \{0,1\}^k$, $row : \{0,1\}^m \to \{0,1\}^n$ and $col : \{0,1\}^m \to \{0,1\}^c$.

The jagged function $p : \{0,1\}^{k+n+c} \to \mathbb{F}$ is now represented by the dense $q : \{0,1\}^m \to \mathbb{F}$ defined as $q(i) = p(tab(i), row(i), col(i))$. The commitment is a dense commitment to q together with the sequence $(t_y, c_y)_{y\in\{0,1\}^k}$ which is sent explicitly.

As in Section 3, the goal is to give an interactive protocol that reduces an evaluation claim on $\hat{p}$ to one on $\hat{q}$.

Evaluation Proofs. Fix table heights $t = (t_y)_{y\in\{0,1\}^k}$ and columns counts $c = (c_y)_{y\in\{0,1\}^k}$. Suppose we are given an evaluation claim of the form $\hat{p}(z_{tab}, z_{row}, z_{ccol}) = v$, where $z_{tab} \in \mathbb{F}^k$, $z_{row} \in \mathbb{F}^n$ and $z_{ccol} \in \mathbb{F}^c$.

Similarly to Section 3, observe that:

$$\hat{p}(z_{tab}, z_{row}, z_{col}) = \sum_{i\in\{0,1\}^m} \hat{q}(i) \cdot \hat{f}_{t,c}(z_{tab}, z_{row}, z_{col}, i), \tag{8}$$

where now $f_{t,c}(z_{tab}, z_r, , z_c, i, t_1, \ldots, t_k, c_1, \ldots, c_k)$ is a function that, given the configuration (t, c), outputs 1 if i is mapped via the natural bijection to (z_r, z_{tab}, z_c), and otherwise outputs 0.

To check the claim, as expected, we employ the sumcheck protocol (Lemma 2.3). To run the sumcheck, the prover needs to first generate the sequence $\left(f_{t,c}(z_{tab}, z_{row}, z_{col}, i)\right)_{i\in\{0,1\}^m}$ which, similarly to Section 3, can be computed by first generating the relevant eq evaluations via Proposition 2.2.

On the verifier's side, the result of the sumcheck is one claim on $\hat{q}$ at a point $i \in \mathbb{F}^m$ and one on $\hat{f}(i)$. The former is the output of the protocol. We show that, also in this case, the verifier can compute the latter on its own, or employ a similar "jagged assist" sub-protocol.

For the verifier to compute $\hat{f}$, following the observation above, we observe that

$$f_{t,c}(z_{tab}, z_{row}, z_{ccol}, i) = \sum_{y\in\{0,1\}^k} eq(z_{tab}, y) \sum_u eq(u, c_y) \cdot \hat{g}_u(z_{row}, z_{col}, i, t_y, t_{y-1})$$

where $g(z_{row}, z_{col}, i, c_y, t_y, t_{y-1})$ is a function that checks that $i = t_{y-1} + z_{row} \cdot 2^u + z_{col}$ and $i < t_y$.

We show that g can be computed by a small-width read-once branching program. Thus, by Lemma 4.2, its multilinear extension can be efficiently computed.

In contrast to the branching program of Proposition 3.2, the current branching program has to add 3 bits. Therefore it needs a trit (i.e., a register taking 3 possible values) to keep track of the carry bit. Additionally it needs a single bit to keep track of the inequality. Overall, the width (i.e., the number of possible memory configurations in every layer) is equal to 6.

Acknowledgements

We thank Ariel Gabizon, Dag Arne Osvik and Justin Thaler for useful comments.

References

[AHIV23] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: lightweight sublinear arguments without a trusted setup. *Des. Codes Cryptogr.*, 91(11):3379–3424, 2023. 5

[BCF+25] Martijn Brehm, Binyi Chen, Ben Fisch, Nicolas Resch, Ron D. Rothblum, and Hadas Zeilberger. Blaze: Fast SNARKs from interleaved RAA codes. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4-8, 2025, Proceedings, Part IV*, volume 15604 of *Lecture Notes in Computer Science*, pages 123–152. Springer, 2025. 5

[BCI+23] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for Reed-Solomon codes. *J. ACM*, 70(5):31:1–31:57, 2023. 3

[BDT24] Suyash Bagad, Yuval Domb, and Justin Thaler. The sum-check protocol over fields of small characteristic. Cryptology ePrint Archive, Paper 2024/1046, 2024. https://eprint.iacr.org/2024/1046. 6

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part II*, volume 14005 of *Lecture Notes in Computer Science*, pages 499–530. Springer, 2023. 8, 18

[CFFZ24] Alessandro Chiesa, Elisabetta Fedele, Giacomo Fenzi, and Andrew Zitek-Estrada. A time-space tradeoff for the sumcheck prover. *IACR Cryptol. ePrint Arch.*, page 524, 2024. 10

[DP25] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4-8, 2025, Proceedings, Part IV*, volume 15604 of *Lecture Notes in Computer Science*, pages 93–122. Springer, 2025. 6

[DT24] Quang Dao and Justin Thaler. Constraint-packing and the sum-check protocol over binary tower fields. Cryptology ePrint Archive, Paper 2024/1038, 2024. https://eprint.iacr.org/2024/1038. 10, 11

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for Muggles. *J. ACM*, 62(4):27:1–27:64, 2015. 8, 9, 18

[GLS+23] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part II*, volume 14082 of *Lecture Notes in Computer Science*, pages 193–226. Springer, 2023. 5

[Gru24] Angus Gruen. Some improvements for the PIOP for zerocheck. *IACR Cryptol. ePrint Arch.*, page 108, 2024. 6, 19

[HR18] Justin Holmgren and Ron Rothblum. Delegating computations with (almost) minimal time and space overhead. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 124–135. IEEE Computer Society, 2018. 8, 14, 15

[Irr24] Irreducible Team. Batched commitment, 2024. https://www.binius.xyz/blueprint/cryptography/commitment/batched/. 9

[LFKN92] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, 1992. 11

[Rot24] Ron D. Rothblum. A note on efficient computation of the multilinear extension. *IACR Cryptol. ePrint Arch.*, page 1103, 2024. 10

[RR24] Noga Ron-Zewi and Ron Rothblum. Local proofs approaching the witness length. *J. ACM*, 71(3):18, 2024. 8, 18

[RRR21] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. *SIAM J. Comput.*, 50(3), 2021. 9

[Set20] Srinath T. V. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 704–737. Springer, 2020. 5, 10

[SR25] Lev Soukhanov and Yaroslav Rebenko. Morgana: a laconic circuit builder. *IACR Cryptol. ePrint Arch.*, page 65, 2025. 10

[ST25] Srinath T. V. Setty and Justin Thaler. Twist and shout: Faster memory checking arguments via one-hot addressing and increments. *IACR Cryptol. ePrint Arch.*, page 105, 2025. 5, 10, 11

[STW24] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Unlocking the lookup singularity with Lasso. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part VI*, volume 14656 of *Lecture Notes in Computer Science*, pages 180–209. Springer, 2024. 5, 10

[VSBW13] Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 223–237. IEEE Computer Society, 2013. 10, 11

[ZCF24] Hadas Zeilberger, Binyi Chen, and Ben Fisch. Basefold: Efficient field-agnostic polynomial commitment schemes from foldable codes. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part X*, volume 14929 of *Lecture Notes in Computer Science*, pages 138–169. Springer, 2024. 3

A Precise Accounting for Sumcheck

In this appendix we give an overview of the exacts cost incurred by the prover in the sumcheck protocol, when restricted to a product of two multilinear polynomials $\hat{f}, \hat{g} : \mathbb{F}^m \to \mathbb{F}$.

The protocol alternates between two procedures:

1. The main one is generating the "round polynomial". In the first round this is $q(x) = \sum_{b_2,\ldots,b_m} f(x, b_2, \ldots, b_m) \cdot g(x, b_3, \ldots, b_m)$. As this polynomial has degree 2, it can be computed by evaluating the above expression at 3 points. However, given the claim that the overall sum is equal to v, it suffices to provide the value of q at the point 0, and the value at 1 is implied by that. Computing $q(0)$ directly takes 2^{m-1} multiplications and $O(2^m)$ additions.

   The additional point $\lambda \in \mathbb{F}$ at which the prover computes q (which must not be 1) requires an evaluation of $\hat{f}(\lambda, b)$ and $\hat{g}(\lambda, b)$, for every $b \in \{0,1\}^{m-1}$. By choosing λ to be one of the points for which multiplication is a basic operation (see Section 1.1) this does not introduce multiplications. Therefore we only need to do 2^{m-1} multiplications to compute the resulting inner product.

   Total: 2^{m-i+1} multiplications in round i.

2. The second operation is geared toward preparing the multilinear for the next round, and simply fixes one of the variables to the random element sent by the verifier in the previous round. This involves 2^{m-1} linear interpolations for both f and g, where each interpolation can be done using a single multiplication.

   Total: 2^{m-i+1} multiplications in round i.

Summing overall rounds, the total number of multiplications is therefore $\sum_{i=1}^{m} 4 \cdot 2^{m-i} \leq 4 \cdot 2^m$.