

Roping in Lasso

Rasmus Kirk Jakobsen

An accessible guide to Lasso, which enables lookup arguments from much larger tables than previously possible. Lasso is the primary component of Jolt, the SNARK-based virtual machine (zkVM) that proves correct execution for RISC-V programs via large table lookups, drastically reducing complexity and prover costs compared to earlier zkVMs. The document assumes minimal familiarity with the constructions that Lasso builds on, by introducing them within a single reference.

2026-04-07

Contents

1. Introduction	3
2. Prerequisites	4
2.1. Proof Systems	4
2.2. SNARKS	5
2.3. Commitment Schemes	6
2.4. Polynomial Commitment Schemes	6
2.5. The Schwartz-Zippel Lemma	8
3. Sumcheck and Multilinear Extensions	9
3.1. Multilinear Extensions	9
3.2. Sumcheck	10
4. GKR	13
4.1. Combining two claims to one	14
4.2. Completing the protocol	16
4.3. Efficiency	18
4.4. Soundness and Completeness	19
5. Productcheck - A Specialized GKR Protocol	21
5.1. The Protocol	21
5.2. Efficiency	24
5.2.1. Computing \tilde{e}_q in linear time	24
5.2.2. Computing \tilde{W} in linear time	24
5.2.3. Putting it all together	25
6. Spartan	27
6.1. Arithmetizing R1CS	28
6.2. Defining the Sumcheck Polynomials g_1, g_2	29
7. Spark	32
7.1. Offline Memory Checking	33
7.2. Constructing a Sparse Polynomial Commitment Scheme	35
7.3. Putting the Pieces Together	37
7.4. The Spark Protocol	39
8. Lasso	41
8.1. Improved Security Analysis	41
8.2. The Lasso Lookup Argument	42
8.3. Efficiency of the Lasso Lookup Argument	44
8.4. Soundness	46
Bibliography	47

1. Introduction

Lookup arguments allow a prover to convince a verifier that a set of values all appear in some predetermined table, without the verifier inspecting each entry. This is a useful tool; many desirable operations in a SNARK circuit, such as range checks and bitwise operations, are expensive to express as primitive arithmetic constraints but trivial to verify via table lookup. Early lookup arguments required the prover to commit to the full table, limiting practical table sizes to roughly 2^{20} entries.

Lasso[1] is the primary component of Jolt, the SNARK-based virtual machine (zkVM) that proves correct execution for RISC-V programs via large table lookups, drastically reducing complexity and prover costs compared to earlier zkVMs. Lasso’s core contribution was that many tables of interest are *decomposable*, meaning that a lookup into a table of size N can be replaced by c lookups into sub-tables of size $N^{1/c}$.

The lookups into these sub-tables are based on the same machinery that drives Spark, the sparse polynomial commitment scheme, employed by Spartan[2]. With these techniques, Lasso achieves prover costs that scale with the number of lookups k and the sub-table size, rather than the full table size N . This makes lookups into tables as large as 2^{128} concretely feasible.

This document presents the constructions that Lasso builds on, introducing them within this single mostly self-contained reference, before arriving at Lasso itself. We assume knowledge of basic algebra (finite fields, polynomials) and basic familiarity with proof systems. These priors are very briefly discussed in Section 2.

The structure is as follows:

- Section 3 briefly introduces multilinear extensions and the sumcheck protocol, the building blocks underlying all other protocols in this document. If these short expositions are insufficient, the reader is encouraged to consult Justin Thaler’s book[3].
- Section 4 presents the GKR[4] interactive proof for layered arithmetic circuits. The exposition mostly follows Thaler’s book[3], but uses the linear-combination technique from Libra[5] to reduce two claims to one at each layer.
- Section 5 specializes GKR to a binary tree of multiplication gates, yielding an interactive proof for the grand product $y \stackrel{?}{=} \prod_{\mathbf{b} \in \mathbb{B}^{\lg(n)}} w(\mathbf{b})$ with a linear-time prover. This follows from a result in Thaler’s 2013 paper[6].
- Section 6 shows how R1CS satisfiability can be reduced to two rounds of sumcheck, both with linear-time provers, following Spartan[2], assuming there exists an efficient sparse polynomial commitment scheme.
- Section 7 introduces Spark, the sparse polynomial commitment scheme that enables Spartan’s linear prover. Spark uses offline memory checking[7] to prove that the prover read the sparse matrix entries honestly.
- Section 8 finally presents the Lasso lookup argument itself.

If the reader is already comfortable with sumcheck, multilinear extensions and the GKR protocol, feel free to start from Section 6.

2. Prerequisites

Throughout this document we use the following notation:

Notation	Meaning
$a \in \mathbb{F}$	A field element of an unspecified field \mathbb{F} .
$[a_1, \dots, a_n] = \mathbf{a} \in \mathbb{F}^n$	A vector of length n consisting of elements from \mathbb{F} .
$a \in_R S$	A value randomly sampled from set S .
$\mathbf{a} \parallel \mathbf{b}$ where $\mathbf{a} \in S^n, \mathbf{b} \in S^m$	Concatenate \mathbf{a}, \mathbf{b} to create a vector $\mathbf{c} \in S^{n+m}$.
$\mathbf{a} \parallel b$ where $\mathbf{a} \in S^n, b \in S$	Concatenate \mathbf{a}, b to create a vector $\mathbf{c} \in S^{n+1}$.
$\mathbb{B} = \{0, 1\}$	A boolean or bit.
$f(x_1, \dots, x_n) = f(\mathbf{x})$	A multivariate polynomial with n variables.
\tilde{f}	A multilinear extension of the function f .
$\hat{f} : \mathbb{B}^n \rightarrow \mathbb{F}$	A lookup table.

2.1. Proof Systems

An Interactive Proof System consists of two Interactive Turing Machines: a computationally unbounded Prover, \mathcal{P} , and a polynomial-time bounded Verifier, \mathcal{V} . The Prover tries to convince the Verifier of a statement $X \in L$, with language L in NP. The following properties must be true:

- **Completeness:** $\forall \mathcal{P} \in \text{ITM}, X \in L \implies \Pr[\langle \mathcal{P}, \mathcal{V} \rangle = \perp] \leq \varepsilon(X)$

For all honest provers, \mathcal{P} , where X is true, the probability that the verifier remains unconvinced ($\langle \mathcal{P}, \mathcal{V} \rangle = \perp$) is negligible in the length of X .

- **Soundness:** $\forall \mathcal{P}^* \in \text{ITM}, X \notin L \implies \Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle = \top] \leq \varepsilon(X)$

For all provers, honest or otherwise, \mathcal{P}^* , that try to convince the verifier of a claim, X , that is not true, the probability that the verifier will be convinced ($\langle \mathcal{P}^*, \mathcal{V} \rangle = \top$) is negligible in the length of X .

An Interactive Argument is similar, but the honest and malicious prover are now polynomially bounded and receive a Private Auxiliary Input, w , not known by \mathcal{V} . This is such that \mathcal{V} can't just compute the answer themselves. Definitions follow:

- **Completeness:** $\forall \mathcal{P}(w) \in \text{PPT}, X \in L \implies \Pr[\langle \mathcal{P}(w), \mathcal{V} \rangle = \perp] \leq \varepsilon(X)$
- **Soundness:** $\forall \mathcal{P}^* \in \text{PPT}, X \notin L \implies \Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle = \top] \leq \varepsilon(X)$

Proofs of knowledge are another type of Proof System. Here the prover claims to know a *witness*, w , for a statement X . Let $X \in L$ and $W(X)$ be the set of witnesses for X that should be accepted in the proof. This allows us to define the following relation: $\mathcal{R} = \{(X, w) : X \in L, w \in W(X)\}$

A proof of knowledge for relation \mathcal{R} is a two party protocol $(\mathcal{P}, \mathcal{V})$ with the following two properties:

- **Knowledge Completeness:** $\Pr[\langle \mathcal{P}(w), \mathcal{V} \rangle = \top] = 1$, i.e. as in Interactive Proof Systems, after an interaction between the prover and verifier the verifier should be convinced with certainty.

- **Knowledge Soundness:** Loosely speaking, Knowledge Soundness requires the existence of an efficient extractor \mathcal{E} that, when given a possibly malicious prover \mathcal{P}^* as input, can extract a valid witness with probability at least as high as the probability that \mathcal{P}^* convinces the verifier \mathcal{V} .

The above proof systems may be *zero-knowledge*, which in loose terms means that anyone looking at the transcript, that is the interaction between prover and verifier, will not be able to tell the difference between a real transcript and one that is simulated. This ensures that an adversary gains no new information beyond what they could have computed on their own. We now define the property more formally:

- **Zero-Knowledge:** For every PPT verifier V^* , there exists a PPT simulator S such that for every $x \in L$, the transcripts produced by S are indistinguishable from the interaction between any verifier and an honest prover:

$$S(x) \sim \text{View}(\langle \mathcal{P}, \mathcal{V}^* \rangle)$$

Where (\sim) denotes indistinguishability. The flavor of Zero-Knowledge depends on the indistinguishability of the transcripts.

Definition 2.1.1 (Distinguishability) Two distributions D_1, D_2 are said to be:

- *Perfectly indistinguishable* ($\stackrel{P}{\sim}$) if they are identical, meaning no observer can tell them apart:

$$\forall x : \Pr[D_1 = x] = \Pr[D_2 = x]$$

- *Statistically indistinguishable* ($\stackrel{S}{\sim}$) if their statistical distance is negligible, meaning that they may differ, but the difference is vanishingly small, even for an unbounded adversary:

$$\Delta(D_1, D_2) := \frac{1}{2} \sum_x |\Pr[D_1 = x] - \Pr[D_2 = x]| \leq \text{negl}(\lambda)$$

- *Computationally indistinguishable* ($\stackrel{C}{\sim}$) if no probabilistic polynomial-time distinguisher \mathcal{A} can tell them apart with more than negligible advantage, though an unbounded adversary might:

$$|\Pr[\mathcal{A}(D_1) = \top] - \Pr[\mathcal{A}(D_2) = \top]| \leq \text{negl}(\lambda)$$

There are generally three types of Zero-Knowledge:

- **Perfect Zero-Knowledge:** $S(x) \stackrel{P}{\sim} \text{View}(\langle \mathcal{P}, \mathcal{V}^* \rangle)$.
- **Statistical Zero-Knowledge:** $S(x) \stackrel{S}{\sim} \text{View}(\langle \mathcal{P}, \mathcal{V}^* \rangle)$.
- **Computational Zero-Knowledge:** $S(x) \stackrel{C}{\sim} \text{View}(\langle \mathcal{P}, \mathcal{V}^* \rangle)$.

2.2. SNARKS

SNARKs - Succinct Noninteractive Arguments of Knowledge - have seen increased usage due to their application in blockchains and cryptocurrencies. They also typically function as general-purpose proof schemes. This means that, given any solution to an NP-problem, the SNARK prover will produce a proof that they know the solution to said NP-problem. Most SNARKs also allow for zero-knowledge arguments, making them zk-SNARKs.

More concretely, imagine that Alice has today's Sudoku problem $X \in \text{NP}$: She claims to have a solution to this problem, her witness, w , and wants to convince Bob without having to reveal the entire solution. She could then use a SNARK to generate a proof for Bob. To do this she must first encode the Sudoku verifier as a circuit R_X , then let x represent public inputs to the circuit, such as today's Sudoku values/positions, etc, and then give the SNARK prover the public inputs and her witness, $\pi = \text{SNARKPROVER}(R_X, x, w)$.

Finally she sends this proof, π , to Bob along with the public Sudoku verifying circuit, R_X , and he can check the proof and be convinced using the SNARK verifier ($\text{SNARKVERIFIER}(R_X, x, \pi)$).

2.3. Commitment Schemes

A commitment scheme is a cryptographic primitive that allows one to commit to a chosen value while keeping it hidden to others, with the ability to reveal the committed value later. Commitment schemes are designed so that the committing party cannot change the value after they have committed to it, i.e. it is *binding*. The fact that anyone who receives the commitment cannot compute the value from the commitment is called *hiding*. For $C = \text{CM.COMMIT}(m)$

- **Perfect Hiding:** Given C , it is impossible to determine m , no matter your computational power.
- **Computational Hiding:** It is computationally infeasible to determine the value committed, from the commitment.
- **Perfect Binding:** It is impossible to change the value committed to, no matter your computational power.
- **Computational Binding:** It is computationally infeasible to change the value committed to.

To reveal a value one can simply send the value to a party that previously received the commitment, and the receiving party can compute the commitment themselves and compare to the previously received commitment.

Pedersen commitments[8] are an instance of a highly useful type of commitment scheme. This is largely because it's a *homomorphic commitment scheme*. Specifically, Pedersen commitments are additively homomorphic, meaning:

$$\text{CM.COMMIT}(m_1) + \text{CM.COMMIT}(m_2) = \text{CM.COMMIT}(m_1 + m_2)$$

That is, you can add the commitments which corresponds to adding the committed inputs and then committing to the result.

2.4. Polynomial Commitment Schemes

In the SNARK section, general-purpose proof schemes were described. Modern general-purpose (zero-knowledge) proof schemes, such as Sonic[9], Plonk[10], Marlin[11] and of course Spartan[2], commonly use *Polynomial Commitment Schemes* (PCSs) for creating their proofs. This means that different PCSs can be used to get security under weaker or stronger assumptions.

- **KZG PCSs:** Uses a trusted setup, which involves generating a Structured Reference String for the KZG commitment scheme[12]. This would give you a traditional SNARK.
- **Bulletproofs PCSs:** Uses an untrusted setup, which is secure assuming the Discrete Log problem is hard, but the verifier is linear in the circuit size.
- **FRI PCSs:** Also uses an untrusted setup, assumes secure one way functions exist. It has a higher constant overhead than PCSs based on the Discrete Log assumption, but because it only assumes that secure one-way functions exist, you end up with a quantum secure PCS.

A PCS allows a prover to prove to a verifier that a committed polynomial evaluates to a certain value, v , given an evaluation input z . There are four main functions used to prove this:

- $\text{PC.SETUP}(\lambda, D) \rightarrow \text{pp}_{\text{PC}}$

The setup routine. Given security parameter λ in unary and a maximum degree bound D . Creates the public parameters pp_{PC} .

- $\text{PC.COMMIT}(p \in \mathbb{F}^{d'}[X], d \in \mathbb{N}) \rightarrow \text{Commit}$

Commits to a degree- d' polynomial p with degree bound d where $d' \leq d$.

- $\text{PC.OPEN}(p \in \mathbb{F}^{d'}[X], C \in \mathbf{Commit}, d \in \mathbb{N}, z \in \mathbb{F}) \rightarrow \mathbf{EvalProof}$

Creates a proof, $\pi \in \mathbf{EvalProof}$, that the degree d' polynomial p , with commitment C , and degree bound d where $d' \leq d$, evaluated at z gives $v = p(z)$.

- $\text{PC.CHECK}(C \in \mathbf{Commit}, d \in \mathbb{N}, z \in \mathbb{F}, v \in \mathbb{F}, \pi \in \mathbf{EvalProof}) \rightarrow \mathbb{B}$

Checks the proof π that claims that the degree d' polynomial p , with commitment C , and degree bound d where $d' \leq d$, evaluates to $v = p(z)$.

Any NP-problem, $X \in \text{NP}$, with a witness w can be compiled into a circuit R_X . This circuit can then be fed to a general-purpose proof scheme prover \mathcal{P}_X along with the witness and public input $(x, w) \in X$, that creates a proof of the statement $R_X(x, w) = \top$. Simplifying slightly, they typically consist of a series of tuples representing opening proofs:

$$(q_1 = (C_1, d, z_1, v_1, \pi_1), \dots, q_m = (C_m, d, z_m, v_m, \pi_m))$$

These tuples can then be verified using PC.CHECK :

$$\text{PC.CHECK}(C_1, d, z_1, v_1, \pi_1) \stackrel{?}{=} 1 \wedge \dots \wedge \text{PC.CHECK}(C_m, d, z_m, v_m, \pi_m) \stackrel{?}{=} 1$$

Along with some checks that the structure of the underlying polynomials p , that q was created from, satisfies any desired relations associated with the circuit R_X .

Then the verifier \mathcal{V}_X will be convinced that w is a valid witness for X . In this way, a proof of knowledge of a witness for any NP-problem can be represented as a series of PCS evaluation proofs.

A PCS has soundness and completeness properties, as well as a binding property:

Definition 2.4.1 (PCS Completeness) For every maximum degree bound $D = \text{poly}(\lambda) \in \mathbb{N}$ and publicly agreed upon $d \in \mathbb{N}$:

$$\Pr \left[\begin{array}{l} \deg(p) \leq d \leq D, \\ \text{PC.CHECK}(C, d, z, v, \pi) = 1 \end{array} \middle| \begin{array}{l} \text{pp}_{\text{PC}} \leftarrow \text{PC.SETUP}(1^\lambda, D), \\ (p, d, z) \leftarrow \mathcal{A}(\text{pp}_{\text{PC}}), \\ v \leftarrow p(z), \\ C \leftarrow \text{PC.COMMIT}(p, d), \\ \pi \leftarrow \text{PC.OPEN}(p, C, d, z) \end{array} \right] = 1$$

In other words, an honest prover will always convince an honest verifier.

Definition 2.4.2 (Knowledge Soundness) For every maximum degree bound $D = \text{poly}(\lambda) \in \mathbb{N}$, polynomial-size adversary \mathcal{A} and publicly agreed upon d , there exists an efficient extractor \mathcal{E} such that the following holds:

$$\Pr \left[\begin{array}{l} \text{PC.CHECK}(C, d, z, v, \pi) = 1 \\ \downarrow \\ C = \text{PC.COMMIT}(p, d) \\ v = p(z), \deg(p) \leq d \leq D \end{array} \middle| \begin{array}{l} \text{pp}_{\text{PC}} \leftarrow \text{PC.SETUP}(1^\lambda, D) \\ (C, d, z, v, \pi) \leftarrow \mathcal{A}(\text{pp}_{\text{PC}}) \\ (p) \leftarrow \mathcal{E}(\text{pp}_{\text{PC}}) \end{array} \right] \geq 1 - \text{negl}(\lambda)$$

In other words, for any adversary, \mathcal{A} , outputting an instance, the knowledge extractor can recover p such that the following holds: C is a commitment to p , $v = p(z)$, and the degree of p is properly bounded. Note that for this protocol, we have *knowledge soundness*, meaning that \mathcal{A} , must actually have knowledge of p (i.e. the \mathcal{E} can extract it).

Definition 2.4.3 (Binding) For every maximum degree bound $D = \text{poly}(\lambda) \in \mathbb{N}$ and publicly agreed upon d , no polynomial-size adversary \mathcal{A} can find two polynomials s.t:

$$\Pr \left[\begin{array}{c} p_1 \in \mathbb{F}[X]_{\leq d}, p_2 \in \mathbb{F}[X]_{\leq d}, p_1 \neq p_2 \\ \wedge \\ C_1 = C_2 \end{array} \middle| \begin{array}{l} \text{pp}_{\text{PC}} \leftarrow \text{PC.SETUP}(1^\lambda, D) \\ (p_1, p_2, d) \leftarrow \mathcal{A}(\text{pp}_{\text{PC}}) \\ C_1 \leftarrow \text{PC.COMMIT}(p_1, d) \\ C_2 \leftarrow \text{PC.COMMIT}(p_2, d) \end{array} \right] \leq \text{negl}(\lambda).$$

In other words, the adversary cannot change the polynomial that they committed to.

2.5. The Schwartz-Zippel Lemma

The Schwartz-Zippel lemma is commonly used in succinct proof systems to test polynomial identities. Formally it states:

$$\xi \in_R \mathbb{F} : \Pr[p(\xi) = 0 \mid p(X) \neq 0] \leq \frac{d}{|\mathbb{F}|}$$

Meaning that if $p(X)$ is not the zero-polynomial, the evaluation at a uniformly random point from \mathbb{F} , will equal zero with at most $d/|\mathbb{F}|$ probability. This can also be used to check equality between polynomials:

$$\begin{array}{l} \xi \in_R \mathbb{F} \\ r(X) = p(X) - q(X) \\ r(\xi) \stackrel{?}{=} 0 \end{array}$$

Or equivalently:

$$p(\xi) \stackrel{?}{=} q(\xi)$$

Meaning that $p(X) = q(X)$ with probability at least $1 - d/|\mathbb{F}|$.

3. Sumcheck and Multilinear Extensions

3.1. Multilinear Extensions

Multilinear extensions are incredibly useful tools for proof systems. One important aspect is that they allow us to model any function $f : \mathbb{B}^{\lceil \lg(n) \rceil} \rightarrow \mathbb{F}$ from bitstrings to field elements as polynomials. This also includes vectors: we can model a vector with length n , consisting of field elements, using such a function:

$$\forall i \in [1..n] : v_i = f(\text{toBits}(i))$$

Example 3.1.1 Take the vector with length $n = 4$:

$$\mathbf{v} = [8, 1, 2, 8]$$

Then we can model this vector with the function $f : \mathbb{B}^{\lg(n)} \rightarrow \mathbb{F}$:

$$f(\mathbf{00}) = 8, \quad f(\mathbf{01}) = 1, \quad f(\mathbf{10}) = 2, \quad f(\mathbf{11}) = 8$$

In general, given any function $f(\mathbf{x}) : \mathbb{B}^\ell \rightarrow \mathbb{F}$, we can create an extension polynomial $\tilde{f}(\mathbf{x})$ such that $\forall \mathbf{b} \in \mathbb{B}^\ell : \tilde{f}(\mathbf{b}) = f(\mathbf{b})$ using Lagrange interpolation, by summing over the boolean hypercube ($\mathbf{b} \in \mathbb{B}^\ell$):

$$\tilde{f}(\mathbf{x}) := \sum_{\mathbf{b} \in \mathbb{B}^\ell} f(\mathbf{b}) \cdot \tilde{e}_{\mathbf{q}_x}(\mathbf{b})$$

Where:

$$\tilde{e}_{\mathbf{q}_x}(\mathbf{b}) := \prod_{i=1}^{\ell} x_i b_i + (1 - x_i)(1 - b_i)$$

This is presented and proved as Fact 3.5 in Thaler's book[3]. These multilinear extension polynomials allow us to encode functions, and by extension vectors too, as polynomials. Then using Schwartz-Zippel from Section 2.5, we can succinctly prove desirable properties about these functions. All proof systems presented in this document follow this methodology.

Furthermore, such a polynomial extension always has degree at most one in each variable and it is *unique*, a fact that we will use throughout the text. The polynomial $\tilde{f}(\mathbf{x})$ is said to be the multilinear extension (MLE) of $f(\mathbf{x})$ and such an MLE is always denoted using a tilde in this document.

It's clear that evaluating $\tilde{e}_{\mathbf{q}_x}(\mathbf{x}, \mathbf{y})$ naively would take $O(\ell)$ time, and thus, it would take $O(2^\ell \cdot \ell)$ time to evaluate \tilde{f} . If we want to remove this ℓ factor, we can make use of Dynamic Programming, by computing a lookup table of $\tilde{e}_{\mathbf{q}_x}$ in $O(2^\ell)$ time.

Lemma 3.1.1 An evaluation table, $\hat{e}_{\mathbf{q}_x}(\mathbf{b})$, for the equality function $\tilde{e}_{\mathbf{q}_x}(\mathbf{b}) : \mathbb{B}^\ell \rightarrow \mathbb{F}$, with a fixed \mathbf{x} can be computed in time $O(2^\ell)$ using $O(2^\ell)$ space.

Proof. To construct $\hat{e}_{\mathbf{q}_x}(\mathbf{b})$ we can use Dynamic Programming.

$$\begin{aligned} \hat{e}_{\mathbf{q}_x}^{(1)}(\mathbf{b} \in \mathbb{B}^1) &= (x_1 b_1 + (1 - x_1)(1 - b_1)) \\ \hat{e}_{\mathbf{q}_x}^{(k)}(\mathbf{b} \in \mathbb{B}^k) &= \hat{e}_{\mathbf{q}_x}^{(k-1)}(b_1, \dots, b_{k-1}) \cdot (x_k b_k + (1 - x_k)(1 - b_k)) \\ \hat{e}_{\mathbf{q}_x}(\mathbf{b} \in \mathbb{B}^\ell) &= \hat{e}_{\mathbf{q}_x}^{(\ell)}(\mathbf{b}) \end{aligned} \tag{1}$$

We first trivially construct $\hat{e}q_{\mathbf{x}}^1(\mathbf{b})$ in $O(1)$ time. Then, we construct $\hat{e}q_{\mathbf{x}}^k(\mathbf{b})$ for each $k \in [1..\ell]$ using Dynamic Programming, which takes $O(2^k)$ time and space. Finally, we get $\hat{e}q_{\mathbf{x}}(\mathbf{b}) = \hat{e}q_{\mathbf{x}}^{(\ell)}(\mathbf{b})$. \square

Example 3.1.2 We show a small example for $|\mathbf{x}| = \ell = 2$:

$$\begin{aligned} \hat{e}q_{\mathbf{x}}^{(1)}[(0)] &:= x_1 \cdot 0 + (1 - 0)(1 - x_1) = 1 - x_1 \\ \hat{e}q_{\mathbf{x}}^{(1)}[(1)] &:= x_1 \cdot 1 + (1 - 1)(1 - x_1) = x_1 \\ \hat{e}q_{\mathbf{x}}^{(2)}[(0, 0)] &:= \hat{e}q_{\mathbf{x}}^{(1)}[(0)] \cdot (1 - x_2) = (1 - x_1) \cdot (1 - x_2) \\ \hat{e}q_{\mathbf{x}}^{(2)}[(0, 1)] &:= \hat{e}q_{\mathbf{x}}^{(1)}[(0)] \cdot x_2 = (1 - x_1) \cdot x_2 \\ \hat{e}q_{\mathbf{x}}^{(2)}[(1, 0)] &:= \hat{e}q_{\mathbf{x}}^{(1)}[(1)] \cdot (1 - x_2) = x_1 \cdot (1 - x_2) \\ \hat{e}q_{\mathbf{x}}^{(2)}[(1, 1)] &:= \hat{e}q_{\mathbf{x}}^{(1)}[(1)] \cdot x_2 = x_1 \cdot x_2 \end{aligned}$$

Each lookup in $\hat{e}q_{\mathbf{x}}^{(k-1)}$ is constant and computing each new entry in $\hat{e}q_{\mathbf{x}}^{(k)}$ takes constant time. There are 2^k entries in $\hat{e}q_{\mathbf{x}}^{(k)}$, so it takes $O(2^k)$ time to compute the table.

Then we can compute the evaluation of any $\tilde{f}(\mathbf{x})$ by utilizing Lemma 3.1.1^o to get $\hat{e}q_{\mathbf{x}}(\mathbf{b})$ and then computing:

$$\tilde{f}(\mathbf{x}) := \sum_{\mathbf{b} \in \mathbb{B}^\ell} f(\mathbf{b}) \cdot \hat{e}q_{\mathbf{x}}(\mathbf{b})$$

This is done by first computing the evaluation table for $\tilde{e}q_{\mathbf{x}}$ in $O(2^\ell)$ time and space and then looking up each value in the above sum \mathbf{b} in the two tables $\tilde{e}q_{\mathbf{x}}, f$ in constant time. A sum over 2^ℓ constant-time operations takes $O(2^\ell)$ time.

corollary 3.1.1 For any function $f(\mathbf{x}) \in \mathbb{B}^\ell \rightarrow \mathbb{F}$, its multilinear extension $\tilde{f}(\mathbf{x})$ can be computed using $O(2^\ell)$ time and space.

3.2. Sumcheck

The sumcheck protocol is an Interactive Proof where the prover, \mathcal{P} , wishes to convince the verifier, \mathcal{V} , of a statement of the following form:

$$y := \sum_{b_1 \in \mathbb{B}} \sum_{b_2 \in \mathbb{B}} \dots \sum_{b_\ell \in \mathbb{B}} g(b_1, \dots, b_\ell)$$

At a high-level, \mathcal{P} starts by sending the claimed value of $g(\mathbf{x})$. The protocol then proceeds in ℓ rounds, wherein each round a single sum is shaved off the expression. For round one, \mathcal{P} sends a specification of the univariate polynomial g_1 :

$$g_1(X) := \sum_{b_{2..\ell} \in \mathbb{B}^{\ell-1}} g(X, b_{2..\ell})$$

Here “specification” may sound vague, but that’s intentional. The protocol is indifferent to whether \mathcal{P} sends $g_1(x)$ in coefficient or evaluation form. \mathcal{P} can either send $\deg(g_1(x)) + 1$ evaluations of the polynomial or the coefficients of $g_1(x)$ to \mathcal{V} . Then, \mathcal{V} checks that:

$$\begin{aligned}
y &\stackrel{?}{=} g_1(0) + g_1(1) \\
&= \left(\sum_{\mathbf{b}_{2:\ell} \in \mathbb{B}^{\ell-1}} g(0, \mathbf{b}_{2:\ell}) \right) + \left(\sum_{\mathbf{b}_{2:\ell} \in \mathbb{B}^{\ell-1}} g(1, \mathbf{b}_{2:\ell}) \right) \\
&= \sum_{\mathbf{b} \in \mathbb{B}^\ell} g(\mathbf{b})
\end{aligned}$$

Along with a degree check that $\deg(g_1) \stackrel{?}{=} \deg_1(g)$. The rest of the rounds proceed in a similar manner, until the final round, where \mathcal{V} also needs to additionally check that $g_\ell(r_\ell) \stackrel{?}{=} g(\mathbf{r})$.

Soundness and Completeness:

The protocol is both sound and complete, with completeness error of $\delta_c = 0$ and a soundness error of $\delta_s \leq \ell \cdot d / |\mathbb{F}|$. Here d is the degree bound of each univariate polynomial sent in the protocol, i.e. $\forall i \in [1..\ell] : \deg(g_i) \leq d$. A proof can be seen in [3] Proposition 4.1.

Efficiency:

- **Communication Cost:** In each round $\deg_j(g(\mathbf{x}))$ field elements are sent by \mathcal{P} and a single field element is sent by \mathcal{V} . So, $O\left(\sum_{j=1}^{\ell} \deg_j(g(\mathbf{x}))\right)$.
- **Verifier Runtime:** The verifiers runtime is proportional to the communication cost plus an additional evaluation of $g(\mathbf{x})$, so $O\left(\text{Eval}_g + \sum_{j=1}^{\ell} \deg_j(g(\mathbf{x}))\right)$.
- **Provers Runtime:** In each round, the prover must evaluate $g(\mathbf{x})$ at $\deg_j(g) + 1$ points for each $2^{\ell-j}$ term. This gives $O\left(\sum_{j=1}^{\ell} \deg_j(g) \cdot 2^{\ell-j} \cdot T\right)$, where T is the cost of a single evaluation of $g(\mathbf{x})$. Usually $\deg_j(g(\mathbf{x}))$ is bounded by some constant, in which case the cost is $O(2^\ell \cdot T)$, since $\sum_{j=1}^{\ell} 2^{\ell-j} = 2^\ell$.

If the individual degree of $g(\mathbf{x})$ is bounded, and T is constant ($O(1)$) then the prover has a runtime of only $O(2^\ell)$. Unfortunately, T is rarely constant, but in Section 5 we give an example of an IP where this is the case, which lets us build an IP that proves that $y \stackrel{?}{=} \prod_{\mathbf{b} \in \mathbb{B}^{\lg(n)}} w(\mathbf{b})$ with a prover runtime linear in n .

The entire sumcheck protocol can be seen below:

Prover

Verifier

=====**Round 1**=====

$$g_1(X) := \sum_{\mathbf{b}_{2:\ell} \in \mathbb{B}^{\ell-1}} g(X, \mathbf{b}_{2:\ell}) \xrightarrow{y, g_1(X)} y \stackrel{?}{=} g_1(0) + g_1(1)$$

$$\deg(g_1) \stackrel{?}{=} \deg_1(g)$$

$$\xleftarrow{r_1} r_1 \in_R \mathbb{F}$$

=====**Round $j \in [2..\ell - 1]$** =====

$$g_j(X) := \sum_{\mathbf{b}_{j+1:\ell} \in \mathbb{B}^{\ell-j}} g(\mathbf{r}_{1:j-1}, X, \mathbf{b}_{j+1:\ell}) \xrightarrow{g_j(X)} g_{j-1}(\mathbf{r}_{j-1}) \stackrel{?}{=} g_j(0) + g_j(1)$$

$$\deg(g_j) \stackrel{?}{=} \deg_j(g)$$

$$\xleftarrow{r_j} r_j \in_R \mathbb{F}$$

=====**Round ℓ** =====

$$g_\ell(X) := g(\mathbf{r}_{1:\ell-1}, X) \xrightarrow{g_\ell(X)} g_{\ell-1}(\mathbf{r}_{\ell-1}) \stackrel{?}{=} g_\ell(0) + g_\ell(1)$$

$$\deg(g_\ell) \stackrel{?}{=} \deg_\ell(g)$$

$$r_\ell \in_R \mathbb{F}$$

$$g_\ell(\mathbf{r}_\ell) \stackrel{?}{=} g(\mathbf{r})$$

4. GKR

Given a circuit \mathcal{C} , with d layers, n inputs and m outputs, a prover (\mathcal{P}) wishes to prove to a verifier (\mathcal{V}) a specific input $\mathbf{w} \in \mathbb{B}^n$ applied to \mathcal{C} produces some output $\mathbf{y} \in \mathbb{B}^m$. Each layer i has a number of either addition or multiplication gates S_i and for notational purposes we also introduce $s_i := \lg(S_i)$. The size of the circuit is the number of gates $S = \sum_{i=0}^d S_i$. \mathcal{P} can leverage the sumcheck protocol, defined earlier.

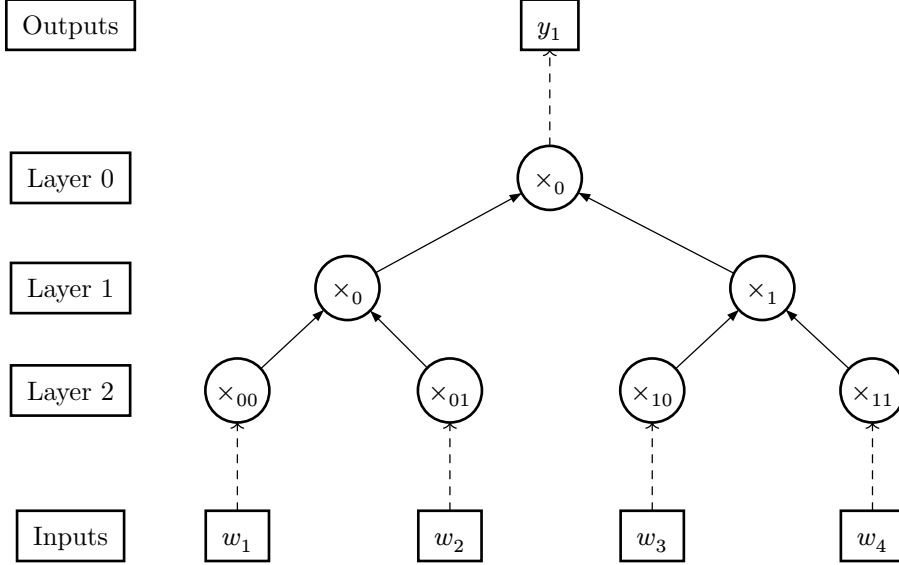


Figure 1: A layered arithmetic circuit for the computation $y_1 = \prod_{i=1}^k w_i$. Each node-label below represents the type of computation and the gate-label for the given layer, so (\times_0) is a multiplication gate with label 0, for layer 0. Note that $d = 3, n = 4, m = 1, S_0 = 1, S_1 = 2, S_2 = 4, S = 7$.

To represent a layered circuit in a form amenable to the sum check protocol, we must first provide an adequate polynomial representation of the circuit. We start with the following three functions:

- $\text{add}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) : \mathbb{B}^{s_i+2s_{i+1}} \rightarrow \mathbb{B}$ which outputs 1 if and only if gate \mathbf{a} is an addition gate and \mathbf{b} is the left input and \mathbf{c} is the right input of gate \mathbf{a} .
- $\text{mul}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) : \mathbb{B}^{s_i+2s_{i+1}} \rightarrow \mathbb{B}$ which outputs 1 if and only if gate \mathbf{a} is a multiplication gate and \mathbf{b} is the left input and \mathbf{c} is the right input of gate \mathbf{a} .
- $W_i(\mathbf{a}) \in \mathbb{B}^{s_i} \rightarrow \mathbb{B}$ which, given the gate-label \mathbf{a} , outputs the value of node \mathbf{a} .

Example 4.1 For Figure 1 $\text{add}_i, \text{mul}_i$ would evaluate to the following values, with $(_)$ denoting any other input:

$$\begin{aligned} \text{mul}_0(\mathbf{0}, \mathbf{0}, \mathbf{0}) &= 1 & \text{mul}_1(\mathbf{0}, \mathbf{00}, \mathbf{01}) &= 1 \\ \text{mul}_0(_) &= 0 & \text{mul}_1(\mathbf{1}, \mathbf{10}, \mathbf{11}) &= 1 \\ \text{add}_0(_) &= 0 & \text{mul}_1(_) &= 0 \\ & & \text{add}_1(_) &= 0 \end{aligned}$$

We can use the multilinear extensions of add_i and mul_i to represent W_i in a form that lets us use sumcheck:

$$\tilde{W}_i(\mathbf{a}) = \sum_{\mathbf{b}, \mathbf{c} \in \mathbb{B}^{s_{i+1}}} \widetilde{\text{add}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) (\tilde{W}_{i+1}(\mathbf{b}) + \tilde{W}_{i+1}(\mathbf{c})) + \widetilde{\text{mul}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot \tilde{W}_{i+1}(\mathbf{b}) \cdot \tilde{W}_{i+1}(\mathbf{c}) \quad (2)$$

Lemma 4.1 The above equation for $\tilde{W}_i(\mathbf{a})$ is indeed the multilinear extension of $W_i(\mathbf{a})$

Proof. In order to prove the Lemma, we need to first prove multilinearity, i.e. that each individual degree of $\widetilde{W}_i(\mathbf{a}) \leq 1$. This is apparent, since even though we multiply polynomials, only the \mathbf{b}, \mathbf{c} variables are affected, not \mathbf{a} .

Next, we need to argue that $\forall \mathbf{a} \in \mathbb{B}^{s_i} : \widetilde{W}_i(\mathbf{a}) = W_i(\mathbf{a})$. This follows from the fact that $\widetilde{\text{add}}_i, \widetilde{\text{mul}}_i, \widetilde{W}_{i+1}$ are all multilinear extensions of their respective functions. We should therefore also be able to rewrite Equation 2° like so:

$$\widetilde{W}_i(\mathbf{a} \in \mathbb{B}^{s_i}) = \sum_{\mathbf{b}, \mathbf{c} \in \mathbb{B}^{s_{i+1}}} \text{add}_i(\mathbf{a}, \mathbf{b}, \mathbf{c})(W_{i+1}(\mathbf{b}) + W_{i+1}(\mathbf{c})) + \text{mul}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot W_{i+1}(\mathbf{b}) \cdot W_{i+1}(\mathbf{c})$$

The definitions of $\text{add}_i, \text{mul}_i$ state that they are only active when the given gate label is an addition or multiplication gate respectively. Thus, for any valid circuit, only one may be active at a time. Additionally, again for any valid circuit, only a single of these terms in the sum will output a value of either:

$$W_{i+1}(\mathbf{b}) + W_{i+1}(\mathbf{c}), \quad W_{i+1}(\mathbf{b}) \cdot W_{i+1}(\mathbf{c})$$

Depending on whether the gate-label refers to an addition or multiplication gate. This is exactly the value that's produced by W_i , as long as \widetilde{W}_{i+1} is indeed the MLE of W_{i+1} . Applying this argument recursively proves the claim. \square

Assume that \mathcal{P} convinces \mathcal{V} that some polynomial $W'(X_1, \dots, X_{s_0}) = W_0(X_1, \dots, X_{s_0})$, then by Lemma 4.1°, W' describes the output of the circuit given the chosen input. Then the verifier can confirm that the evaluations of the circuit given input \mathbf{w} evaluates to \mathbf{y} by simply evaluating $W'(X_1, \dots, X_{s_0})$ on all gate labels in the output layer. To prove $W' = W_0$, \mathcal{V} chooses a uniformly random \mathbf{r} and wishes to verify that $\widetilde{W}'(\mathbf{r}) = \widetilde{W}_0(\mathbf{r})$, which by Schwartz-Zippel means that $\widetilde{W}' = \widetilde{W}_0$, and by definition of MLEs $W' = W_0$. Therefore, \mathcal{P} and verifier apply sumcheck to the following polynomial:

$$f_0(\mathbf{b}, \mathbf{c}) = \widetilde{\text{add}}_0(\mathbf{r}, \mathbf{b}, \mathbf{c})(\widetilde{W}_1(\mathbf{b}) + \widetilde{W}_1(\mathbf{c})) + \widetilde{\text{mul}}_0(\mathbf{r}, \mathbf{b}, \mathbf{c}) \cdot \widetilde{W}_1(\mathbf{b}) \cdot \widetilde{W}_1(\mathbf{c})$$

Which, if successful, convinces \mathcal{V} that $\widetilde{W}'(\mathbf{r}) = \widetilde{W}_0(\mathbf{r}) \implies W' = W_0$ as desired. But in the final round of the sumcheck protocol, \mathcal{V} must be able to evaluate the above polynomial at a random point. The functions $\widetilde{\text{add}}_0$ and $\widetilde{\text{mul}}_0$ are part of the circuit description, and can thus be computed by \mathcal{V} without help from \mathcal{P} ¹. But \mathcal{V} also needs to evaluate \widetilde{W}_1 at two separate random points $\mathbf{b}'_0, \mathbf{c}'_0 \in_R \mathbb{F}^{s_1}$ corresponding to \mathbf{b}, \mathbf{c} :

$$f_0(\mathbf{b}'_0, \mathbf{c}'_0) = \widetilde{\text{add}}_0(\mathbf{r}, \mathbf{b}'_0, \mathbf{c}'_0)(\widetilde{W}_1(\mathbf{b}'_0) + \widetilde{W}_1(\mathbf{c}'_0)) + \widetilde{\text{mul}}_0(\mathbf{r}, \mathbf{b}'_0, \mathbf{c}'_0) \cdot \widetilde{W}_1(\mathbf{b}'_0) \cdot \widetilde{W}_1(\mathbf{c}'_0)$$

In principle, we could run the sumcheck protocol twice, on the polynomials:

$$\begin{aligned} f_1(\mathbf{b}, \mathbf{c}) &= \widetilde{\text{add}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c})(\widetilde{W}_2(\mathbf{b}) + \widetilde{W}_2(\mathbf{c})) + \widetilde{\text{mul}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) \cdot \widetilde{W}_2(\mathbf{b}) \cdot \widetilde{W}_2(\mathbf{c}) \\ f_1(\mathbf{b}, \mathbf{c}) &= \widetilde{\text{add}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c})(\widetilde{W}_2(\mathbf{b}) + \widetilde{W}_2(\mathbf{c})) + \widetilde{\text{mul}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) \cdot \widetilde{W}_2(\mathbf{b}) \cdot \widetilde{W}_2(\mathbf{c}) \end{aligned} \tag{3}$$

But this would result in an exponential amount of sumchecks in the depth d , which would be quite a problem! Instead, we can reduce two checks into one, using a linear combination.

4.1. Combining two claims to one

Suppose we were to apply sumcheck to the following polynomial instead:

$$q(\mathbf{b}'_0, \mathbf{c}'_0) := \widetilde{W}_1(\mathbf{b}'_0) + \alpha \cdot \widetilde{W}_1(\mathbf{c}'_0)$$

¹This can be done by \mathcal{V} in at least $O(S_i)$ time, reducing this to $O(\lg(S_i))$ is important to achieve a sublinear verifier if this IP is turned into an Interactive Argument.

Which can be derived as:

$$\begin{aligned}
q(\mathbf{b}'_0, \mathbf{c}'_0) &= \left(\sum_{\mathbf{b}, \mathbf{c} \in \mathbb{B}^{s_2}} \widetilde{\text{add}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) (\widetilde{W}_2(\mathbf{b}) + \widetilde{W}_2(\mathbf{c})) + \widetilde{\text{mul}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) \cdot \widetilde{W}_2(\mathbf{b}) \cdot \widetilde{W}_2(\mathbf{c}) \right) + \\
&\quad \alpha \cdot \left(\sum_{\mathbf{b}, \mathbf{c} \in \mathbb{B}^{s_2}} \widetilde{\text{add}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) (\widetilde{W}_2(\mathbf{b}) + \widetilde{W}_2(\mathbf{c})) + \widetilde{\text{mul}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) \cdot \widetilde{W}_2(\mathbf{b}) \cdot \widetilde{W}_2(\mathbf{c}) \right) \\
&= \sum_{\mathbf{b}, \mathbf{c} \in \mathbb{B}^{s_2}} \widetilde{\text{add}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) (\widetilde{W}_2(\mathbf{b}) + \widetilde{W}_2(\mathbf{c})) + \widetilde{\text{mul}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) \cdot \widetilde{W}_2(\mathbf{b}) \cdot \widetilde{W}_2(\mathbf{c}) + \\
&\quad \alpha \cdot \widetilde{\text{add}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) (\widetilde{W}_2(\mathbf{b}) + \widetilde{W}_2(\mathbf{c})) + \alpha \cdot \widetilde{\text{mul}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) \cdot \widetilde{W}_2(\mathbf{b}) \cdot \widetilde{W}_2(\mathbf{c}) \\
&= \sum_{\mathbf{b}, \mathbf{c} \in \mathbb{B}^{s_2}} \left(\widetilde{\text{add}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) + \alpha \cdot \widetilde{\text{add}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) \right) (\widetilde{W}_2(\mathbf{b}) + \widetilde{W}_2(\mathbf{c})) + \\
&\quad \left(\widetilde{\text{mul}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) + \alpha \cdot \widetilde{\text{mul}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) \right) (\widetilde{W}_2(\mathbf{b}) \cdot \widetilde{W}_2(\mathbf{c}))
\end{aligned} \tag{4}$$

The Lemma² below shows how this will help the prover-verifier pair in showing that $v_{\mathbf{b}'_0} = \widetilde{W}_1(\mathbf{b}'_0) \wedge v_{\mathbf{c}'_0} = \widetilde{W}_1(\mathbf{c}'_0)$, thus enabling \mathcal{V} to compute $f_0(\mathbf{b}'_0, \mathbf{c}'_0)$:

Lemma 4.1.1 For a polynomial $p(X_1, \dots, X_\ell)$, if a prover wants to convince a verifier of two claims $v_1 = p(\mathbf{r}_1), v_2 = p(\mathbf{r}_2)$, then they can reduce this to a single claim over a polynomial $q(\mathbf{r}_1, \mathbf{r}_2)$, using a uniformly random $\alpha \in_R \mathbb{F}$:

$$q(X_1, \dots, X_{2\ell}) := p(X_1, \dots, X_\ell) + \alpha \cdot p(X_{\ell+1}, \dots, X_{2\ell})$$

\mathcal{V} can then check that $q(\mathbf{r}_1, \mathbf{r}_2) = v_1 + \alpha \cdot v_2$. The claim that $v_1 = p(\mathbf{r}_1) \wedge v_2 = p(\mathbf{r}_2)$ will then hold except with probability of $\frac{1}{|\mathbb{F}|}$, given that $q(X_1, \dots, X_{2\ell})$ is defined as above.

Proof. If $q(\mathbf{r}_1, \mathbf{r}_2) = p(\mathbf{r}_1) + \alpha \cdot p(\mathbf{r}_2)$ but the claim does not hold, i.e. $v_1 \neq p(\mathbf{r}_1) \vee v_2 \neq p(\mathbf{r}_2)$ then that would mean that the following univariate polynomial is a non-zero polynomial:

$$e(X) = v_1 + X \cdot v_2 - (p(\mathbf{r}_1) + X \cdot p(\mathbf{r}_2))$$

And that it still evaluated to zero, which by the Schwartz-Zippel Lemma, has probability:

$$\Pr[e(\alpha) = 0 \mid e(X) \neq 0] = \frac{\deg(e)}{|\mathbb{F}|} = \frac{1}{|\mathbb{F}|}$$

□

In the GKR protocol, running sumcheck on Equation 4^o convinces \mathcal{V} that $q(\mathbf{b}'_0, \mathbf{c}'_0) = \widetilde{W}_1(\mathbf{b}'_0) + \alpha \cdot \widetilde{W}_1(\mathbf{c}'_0)$, which means that \mathcal{V} knows that q is defined as in Lemma 4.1.1^o and they know the evaluation of q at $\mathbf{b}'_0, \mathbf{c}'_0$. \mathcal{V} can then verify that $v_{\mathbf{b}'_0} = \widetilde{W}_1(\mathbf{b}'_0)$ and $v_{\mathbf{c}'_0} = \widetilde{W}_1(\mathbf{c}'_0)$ by additionally checking that $q(\mathbf{b}'_0, \mathbf{c}'_0) = v_{\mathbf{b}'_0} + \alpha \cdot v_{\mathbf{c}'_0}$.

With $v_{\mathbf{b}'_0}$ and $v_{\mathbf{c}'_0}$ \mathcal{V} can compute the evaluation of $f_0(\mathbf{b}'_0, \mathbf{c}'_0)$:

$$f_0(\mathbf{b}'_0, \mathbf{c}'_0) = \widetilde{\text{add}}_0(\mathbf{r}, \mathbf{b}'_0, \mathbf{c}'_0) (v_{\mathbf{b}'_0} + v_{\mathbf{c}'_0}) + \widetilde{\text{mul}}_0(\mathbf{r}, \mathbf{b}'_0, \mathbf{c}'_0) \cdot v_{\mathbf{b}'_0} \cdot v_{\mathbf{c}'_0}$$

Thus, when we proceed to layer one, the polynomial we do sumcheck on would be:

²This approach is also used in Libra[5], but they don't present a soundness proof since they deem it "trivial". The Lemma in this document also uses a single random variable rather than two.

$$f_1(\mathbf{b}, \mathbf{c}) := \left(\widetilde{\text{add}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) + \alpha \cdot \widetilde{\text{add}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) \right) \left(\widetilde{W}_2(\mathbf{b}) + \widetilde{W}_2(\mathbf{c}) \right) + \\ \left(\widetilde{\text{mul}}_1(\mathbf{b}'_0, \mathbf{b}, \mathbf{c}) + \alpha \cdot \widetilde{\text{mul}}_1(\mathbf{c}'_0, \mathbf{b}, \mathbf{c}) \right) \left(\widetilde{W}_2(\mathbf{b}) \cdot \widetilde{W}_2(\mathbf{c}) \right)$$

It should already now be apparent that we can repeat this procedure, all the way to the input layer d .

4.2. Completing the protocol

In the layer before the input layer ($d - 1$), the final check in the sumcheck protocol requires \mathcal{V} to evaluate the following polynomial at $\mathbf{b}'_{d-1}, \mathbf{c}'_{d-1}$:

$$f_{d-1}(\mathbf{b}, \mathbf{c}) := \left(\widetilde{\text{add}}_{d-1}(\mathbf{b}'_{d-2}, \mathbf{b}, \mathbf{c}) + \alpha \cdot \widetilde{\text{add}}_{d-1}(\mathbf{c}'_{d-2}, \mathbf{b}, \mathbf{c}) \right) \left(\widetilde{W}_d(\mathbf{b}) + \widetilde{W}_d(\mathbf{c}) \right) + \\ \left(\widetilde{\text{mul}}_{d-1}(\mathbf{b}'_{d-2}, \mathbf{b}, \mathbf{c}) + \alpha \cdot \widetilde{\text{mul}}_{d-1}(\mathbf{c}'_{d-2}, \mathbf{b}, \mathbf{c}) \right) \left(\widetilde{W}_d(\mathbf{b}) \cdot \widetilde{W}_d(\mathbf{c}) \right)$$

The polynomials $\widetilde{\text{add}}_{d-1}$ and $\widetilde{\text{mul}}_{d-1}$ can be evaluated as usual. The polynomial \widetilde{W}_d corresponds to the values of the input layer \mathbf{w} . Since \mathcal{V} knows \mathbf{w} they can compute the multilinear extension of \mathbf{w} corresponding to W_d . From this the verifier can compute $\widetilde{W}_d(\mathbf{b}'_d), \widetilde{W}_d(\mathbf{c}'_d)$ and thus the evaluation of $f_{d-1}(\mathbf{b}'_{d-1}, \mathbf{c}'_{d-1})$.

The entire protocol can be seen on the next page:

$$\hat{x} := (\text{add}_0, \dots, \text{add}_d, \text{mul}_0, \dots, \text{mul}_d)$$

Prover(\hat{x}, w)

Verifier(\hat{x}, w)

===== **Preprocessing** =====

\mathcal{P} sends W' to \mathcal{V} claiming that $W' = W_0$. \mathcal{V} then picks out a random r . After this point, \mathcal{P} wants to prove that $m_0 = \tilde{W}'_0(r)$.

$$W' : \mathbb{B}^{s_0} \rightarrow \mathbb{F} \xrightarrow{W'} r \in_R \mathbb{F}^{s_0}, m_0 := \tilde{W}'_0(r)$$

===== **Round 0** =====

\mathcal{P} defines the $(2s_i)$ -variate polynomial, $f_0(\mathbf{b}, \mathbf{c})$, \mathcal{P} and \mathcal{V} then perform sumcheck on the polynomial:

$$f_0(\mathbf{b}, \mathbf{c}) = \widetilde{\text{add}}_0(r, \mathbf{b}, \mathbf{c}) (\tilde{W}'_1(\mathbf{b}) + \tilde{W}'_1(\mathbf{c})) + \widetilde{\text{mul}}_0(r, \mathbf{b}, \mathbf{c}) \cdot \tilde{W}'_1(\mathbf{b}) \cdot \tilde{W}'_1(\mathbf{c})$$

$$\sum_{\mathbf{b}, \mathbf{c} \in \mathbb{B}^{s_1}} f_0(\mathbf{b}, \mathbf{c}) \stackrel{?}{=} m_0$$

At the end of the protocol, \mathcal{P} sends \mathcal{V} the evaluations of $v_{\mathbf{b}'_0} := \tilde{W}'_1(\mathbf{b}'_0)$ and $v_{\mathbf{c}'_0} := \tilde{W}'_1(\mathbf{c}'_0)$, so \mathcal{V} can make the final check in the sumcheck protocol.

$$v_{\mathbf{b}'_0} := \tilde{W}'_1(\mathbf{b}'_0), v_{\mathbf{c}'_0} := \tilde{W}'_1(\mathbf{c}'_0) \xrightarrow{v_{\mathbf{b}'_0}, v_{\mathbf{c}'_0}} m_0 \stackrel{?}{=} \widetilde{\text{add}}_0(r, \mathbf{b}'_0, \mathbf{c}'_0) \cdot (v_{\mathbf{b}'_0} + v_{\mathbf{c}'_0}) + \widetilde{\text{mul}}_0(r, \mathbf{b}'_0, \mathbf{c}'_0) \cdot v_{\mathbf{b}'_0} \cdot v_{\mathbf{c}'_0}$$

===== **Round $i \in [1..d-1]$** =====

To verify the values $v_{\mathbf{b}'_{i-1}}, v_{\mathbf{c}'_{i-1}}$ from the previous round, \mathcal{V} picks out a randomly sampled value α and sends it to \mathcal{P} . \mathcal{P} uses α to construct the combined-claim polynomial, for which \mathcal{P} and \mathcal{V} perform the sumcheck over:

$$f_i(\mathbf{b}, \mathbf{c}) := (\widetilde{\text{add}}_i(\mathbf{b}'_{i-1}, \mathbf{b}, \mathbf{c}) + \alpha \cdot \widetilde{\text{add}}_i(\mathbf{c}'_{i-1}, \mathbf{b}, \mathbf{c})) (\tilde{W}'_{i+1}(\mathbf{b}) + \tilde{W}'_{i+1}(\mathbf{c})) +$$

$$(\widetilde{\text{mul}}_i(\mathbf{b}'_{i-1}, \mathbf{b}, \mathbf{c}) + \alpha \cdot \widetilde{\text{mul}}_i(\mathbf{c}'_{i-1}, \mathbf{b}, \mathbf{c})) (\tilde{W}'_{i+1}(\mathbf{b}) \cdot \tilde{W}'_{i+1}(\mathbf{c}))$$

$$f_i(\mathbf{b}, \mathbf{c}) \xleftarrow{\alpha} \alpha \in_R \mathbb{F}$$

$$\sum_{\mathbf{b}, \mathbf{c} \in \mathbb{B}^{s_{i+1}}} f_i(\mathbf{b}, \mathbf{c}) \stackrel{?}{=} m_i$$

At the end of the protocol, \mathcal{P} sends \mathcal{V} the evaluations of $v_{\mathbf{b}'_i} := \tilde{W}'_{i+1}(\mathbf{b}'_i)$ and $v_{\mathbf{c}'_i} := \tilde{W}'_{i+1}(\mathbf{c}'_i)$, so \mathcal{V} can make the final check in the sumcheck protocol:

$$m'_i := (\widetilde{\text{add}}_i(\mathbf{b}'_{i-1}, \mathbf{b}'_i, \mathbf{c}'_i) + \alpha \cdot \widetilde{\text{add}}_i(\mathbf{c}'_{i-1}, \mathbf{b}'_i, \mathbf{c}'_i)) (v_{\mathbf{b}'_i} + v_{\mathbf{c}'_i}) +$$

$$(\widetilde{\text{mul}}_i(\mathbf{b}'_{i-1}, \mathbf{b}'_i, \mathbf{c}'_i) + \alpha \cdot \widetilde{\text{mul}}_i(\mathbf{c}'_{i-1}, \mathbf{b}'_i, \mathbf{c}'_i)) (v_{\mathbf{b}'_i} \cdot v_{\mathbf{c}'_i})$$

$$v_{\mathbf{b}'_i} := \tilde{W}'_{i+1}(\mathbf{b}'_i), v_{\mathbf{c}'_i} := \tilde{W}'_{i+1}(\mathbf{c}'_i) \xrightarrow{v_{\mathbf{b}'_i}, v_{\mathbf{c}'_i}} m'_i \stackrel{?}{=} m_i$$

===== **Round d** =====

At the input layer d , \mathcal{V} has two claims $v_{\mathbf{b}'_{d-1}}$ and $v_{\mathbf{c}'_{d-1}}$. \mathcal{V} constructs \tilde{W}'_d from

$$w. \mathcal{V} \text{ then finally checks that } \tilde{W}'_d(\mathbf{b}'_{d-1}) \stackrel{?}{=} v_{\mathbf{b}'_{d-1}} \text{ and } \tilde{W}'_d(\mathbf{c}'_{d-1}) \stackrel{?}{=} v_{\mathbf{c}'_{d-1}} \wedge$$

$$\tilde{W}'_d(\mathbf{b}'_{d-1}) \stackrel{?}{=} v_{\mathbf{b}'_{d-1}} \wedge \tilde{W}'_d(\mathbf{c}'_{d-1}) \stackrel{?}{=} v_{\mathbf{c}'_{d-1}}$$

4.3. Efficiency

Communication: $O(S_0 + d \cdot \lg(S))$

The GKR protocol has the following communication costs:

- **Preprocessing:** W' , which has size $2^{s_0} = S_0$.
- **Round 0:** $O(s_1)$
 - One sumcheck: $O\left(\sum_{j=1}^{2^{s_1}} \deg_j(f_0)\right)$, and since $\deg_j(f_0)$ is constant; $O(s_1)$
 - Two evaluations: $v_{b'_0}, v_{c'_0}$
- **Round i :** $O(s_{i+1})$
 - One challenge: α
 - One sumcheck: $O\left(\sum_{j=1}^{2^{s_{i+1}}} \deg_j(f_i)\right)$, and since $\deg_j(f_i)$ is constant; $O(s_{i+1})$
 - Two evaluations: $v_{b'_i}, v_{c'_i}$

For a total of $O\left(S_0 + \sum_{i=0}^{d-1} s_{i+1}\right) = O\left(S_0 + \sum_{i=0}^{d-1} \lg(S_i)\right) = O(S_0 + d \cdot \lg(S))$ communication.

Evaluating $\widetilde{\text{add}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}), \widetilde{\text{mul}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c})$ in linear time:

Before discussing the runtime cost of \mathcal{P} and \mathcal{V} , we first describe how $\widetilde{\text{add}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and $\widetilde{\text{mul}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c})$ can be evaluated in $O(S_i)$ time.

First we define two functions $\text{in}_L^{(i)}(\mathbf{a}), \text{in}_R^{(i)}(\mathbf{a}) \in \mathbb{B}^{s_i} \rightarrow \mathbb{B}^{s_{i+1}}$ which, when given a node-label returns the left and right incoming node's labels respectively. We also define the functions $\text{isAdd}(\mathbf{a}), \text{isMul}(\mathbf{a}) \in \mathbb{B}^{s_i} \rightarrow \mathbb{B}$, that define whether \mathbf{a} is an addition or multiplication gate respectively. Then we can define:

$$\begin{aligned} \text{add}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) &:= \text{isAdd}(\mathbf{a}) \wedge \mathbf{b} \stackrel{?}{=} \text{in}_L^{(i)}(\mathbf{a}) \wedge \mathbf{c} \stackrel{?}{=} \text{in}_R^{(i)}(\mathbf{a}) \\ \text{mul}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) &:= \text{isMul}(\mathbf{a}) \wedge \mathbf{b} \stackrel{?}{=} \text{in}_L^{(i)}(\mathbf{a}) \wedge \mathbf{c} \stackrel{?}{=} \text{in}_R^{(i)}(\mathbf{a}) \end{aligned}$$

And their MLEs:

$$\begin{aligned} \widetilde{\text{add}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) &:= \sum_{\mathbf{a}' \in \mathbb{B}^{s_i}} \tilde{\text{eq}}_{\mathbf{a}}(\mathbf{a}') \cdot \text{isAdd}_i(\mathbf{a}') \cdot \tilde{\text{eq}}_{\mathbf{b}}\left(\text{in}_L^{(i)}(\mathbf{a}')\right) \cdot \tilde{\text{eq}}_{\mathbf{c}}\left(\text{in}_R^{(i)}(\mathbf{a}')\right) \\ \widetilde{\text{mul}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) &:= \sum_{\mathbf{a}' \in \mathbb{B}^{s_i}} \tilde{\text{eq}}_{\mathbf{a}}(\mathbf{a}') \cdot \text{isMul}_i(\mathbf{a}') \cdot \tilde{\text{eq}}_{\mathbf{b}}\left(\text{in}_L^{(i)}(\mathbf{a}')\right) \cdot \tilde{\text{eq}}_{\mathbf{c}}\left(\text{in}_R^{(i)}(\mathbf{a}')\right) \end{aligned}$$

Upon evaluation, we can create a table for each $\tilde{\text{eq}}_{\mathbf{a}}(\mathbf{a}'), \tilde{\text{eq}}_{\mathbf{b}}\left(\text{in}_L^{(i)}(\mathbf{a}')\right), \tilde{\text{eq}}_{\mathbf{c}}\left(\text{in}_R^{(i)}(\mathbf{a}')\right)$ in $O(3 \cdot 2^{s_i}) = O(S_i)$ time, using Lemma 3.1.1°. Thus, an evaluation of $\widetilde{\text{add}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c})$ or $\widetilde{\text{mul}}_i(\mathbf{a}, \mathbf{b}, \mathbf{c})$ is bounded by $O(S_i)$ time.

Verifier Runtime: $O(S_0 + d \cdot \lg(S) + S + n)$

In each round, \mathcal{V} needs to do work roughly proportional to the communication complexity, i.e. $O(d \cdot \lg(S))$ across all rounds. Note that \mathcal{V} does not evaluate each sumcheck polynomial fully in the final round of sumcheck, as $v_{b'_i}, v_{c'_i}$ are provided by \mathcal{P} . \mathcal{V} only needs to evaluate $\widetilde{\text{add}}(\mathbf{a}, \mathbf{b}, \mathbf{c}), \widetilde{\text{mul}}(\mathbf{a}, \mathbf{b}, \mathbf{c})$ in each round. Let's say *all* these evaluations throughout the protocol costs t . If $\widetilde{\text{add}}, \widetilde{\text{mul}}$ has no special structure for \mathcal{V} to exploit, we can set $t = O\left(\sum_{i=0}^d S_i\right) = O(S)$ following the discussion above. In the final round, \mathcal{V} needs to evaluate $\widetilde{W}_d(\mathbf{b}'_{d-1}), \widetilde{W}_d(\mathbf{c}'_{d-1})$. Both evaluations cost $2^{s_d} = n$ by Lemma 3.1.1°. Then we get a verifier runtime of $O(S_0 + d \cdot \lg(S) + S + n)$.

Prover Runtime: $O(S^3)$

For each round of the GKR protocol, the dominant cost for \mathcal{P} will always be the sumcheck which was stated to be $O(2^\ell \cdot T)$ in Section 3.2. Where ℓ is the number of variables in each sumcheck polynomial ($2s_{i+1}$) and T is the cost to evaluate the sumcheck polynomial at a single point. In order to evaluate the sumcheck

polynomial, \mathcal{P} needs to evaluate $\widetilde{\text{add}}_i(\mathbf{a}'_{i-1}, \mathbf{b}, \mathbf{c}), \widetilde{\text{mul}}_i(\mathbf{a}'_{i-1}, \mathbf{b}, \mathbf{c}), \widetilde{W}_i(\mathbf{b})$ and $\widetilde{W}_i(\mathbf{b})$. It was argued earlier that $\widetilde{\text{add}}_i, \widetilde{\text{mul}}_i$ could be evaluated in $O(S_i)$ time. The polynomial \widetilde{W}_i can be evaluated in $O(S_{i+1})$ using Lemma 3.1.1^o, totaling $O(S_i + S_{i+1})$ to evaluate the sumcheck polynomial at a single point.

Then \mathcal{P} 's runtime for each round in the GKR protocol is:

$$\begin{aligned} O(2^\ell \cdot T) &= O(2^{2s_{i+1}} \cdot T) \\ &= O(2^{s_{i+1}} \cdot 2^{s_{i+1}} \cdot T) \\ &= O(S_{i+1} \cdot S_{i+1} \cdot (S_i + S_{i+1})) \\ &= O(S_{i+1}^3 + S_i S_{i+1}^2) \end{aligned}$$

And the total runtime of \mathcal{P} is bounded by:

$$\begin{aligned} \sum_{i=0}^{d-1} O(S_i S_{i+1}^2 + S_{i+1}^3) &= O\left(\sum_{i=0}^{d-1} (S_i S_{i+1}^2 + S_{i+1}^3)\right) \\ &\leq O\left(\left(\sum_{i=0}^{d-1} S_{i+1}\right)^3 + \left(\sum_{i=0}^{d-1} S_{i+1}\right)^2 \left(\sum_{i=0}^{d-1} S_i\right)\right) \\ &= O\left(\left(\sum_{i=1}^d S_i\right)^3 + \left(\sum_{i=1}^d S_i\right)^2 \left(\sum_{i=0}^{d-1} S_i\right)\right) \\ &= O\left(\left(\sum_{i=0}^d S_i\right)^3 + \left(\sum_{i=0}^d S_i\right)^2 \left(\sum_{i=0}^d S_i\right)\right) \\ &= O(S^3) \end{aligned}$$

4.4. Soundness and Completeness

Given Lemma 4.1^o, Lemma 4.1.1^o and the completeness of the sumcheck protocol, it's easy to see that the GKR protocol is complete.

As for soundness, assume that \mathcal{P} wants to cheat, and wishes to convince \mathcal{V} that the output of \mathcal{C} when given input \mathbf{w} is \mathbf{y}' , when it's actually \mathbf{y} , i.e. $\mathbf{y}' \neq \mathbf{y}$ but the verifier still accepts. This must then mean that $W' \neq W_0$.

Schwartz-Zippel states that the probability of $e(\mathbf{x}) = \widetilde{W}'(\mathbf{x}) - \widetilde{W}_0(\mathbf{x})$ evaluating to zero for a random input r is bounded by $\frac{\deg(E)}{|\mathbb{F}|} = \frac{s_0}{|\mathbb{F}|}$. Let's call this event $E_{W'}$.

For each round in the sumcheck protocol \mathcal{P} sends g_j and the verifier checks that $g_j(r_{j-1}) \stackrel{?}{=} g_j(0) + g_j(1)$. This has a $\frac{\deg(g_j)}{|\mathbb{F}|}$ probability of cheating \mathcal{V} by Schwartz-Zippel. The individual degree of g_j is at most two, so we can write; $\frac{2}{|\mathbb{F}|}$. Call this event E_j .

For round zero in the GKR protocol:

$$m_0 \stackrel{?}{=} \widetilde{\text{add}}_0(\mathbf{r}, \mathbf{b}'_0, \mathbf{c}'_0) \cdot (v_{\mathbf{b}'_0} + v_{\mathbf{c}'_0}) + \widetilde{\text{mul}}_0(\mathbf{r}, \mathbf{b}'_0, \mathbf{c}'_0) \cdot v_{\mathbf{b}'_0} \cdot v_{\mathbf{c}'_0}$$

Since all this computation (including m_0) is computed by \mathcal{V} , \mathcal{P} can't cheat here, except in each round of the sumcheck, which has already been covered. This assumes that $v_{\mathbf{b}'_0}, v_{\mathbf{c}'_0}$ are indeed the correct evaluations, which is verified in the next round.

For each GKR round $i \in [1..d-1]$, the probability that the check:

$$m_i \stackrel{?}{=} \left(\widetilde{\text{add}}_i(\mathbf{b}'_{i-1}, \mathbf{b}'_i, \mathbf{c}'_i) + \alpha \cdot \widetilde{\text{add}}_i(\mathbf{c}'_{i-1}, \mathbf{b}'_i, \mathbf{c}'_i) \right) (v_{\mathbf{b}'_i} + v_{\mathbf{c}'_i}) + \\ \left(\widetilde{\text{mul}}_i(\mathbf{b}'_{i-1}, \mathbf{b}'_i, \mathbf{c}'_i) + \alpha \cdot \widetilde{\text{mul}}_i(\mathbf{c}'_{i-1}, \mathbf{b}'_i, \mathbf{c}'_i) \right) (v_{\mathbf{b}'_i} \cdot v_{\mathbf{c}'_i})$$

Passing without the statement holding is $\frac{1}{|\mathbb{F}|}$ by Lemma 4.1.1^o, again provided that $v_{\mathbf{b}'_i}, v_{\mathbf{c}'_i}$ are indeed the correct evaluations. Call this event E_i .

\mathcal{P} can't cheat in the final round, as they cannot affect the verifier's check.

We can union-bound these events to get a soundness bound.

$$\begin{aligned} \delta_s &= \Pr[E_{W'}] + \bigcup_{i=0}^{d-1} \bigcup_{j=0}^{s_i} \Pr[E_j] + \bigcup_{i=1}^{d-1} \Pr[E_i] \\ &\leq \frac{s_0}{|\mathbb{F}|} + \sum_{i=0}^{d-1} \sum_{j=0}^{2s_{i+1}} \frac{2}{|\mathbb{F}|} + \sum_{i=1}^{d-1} \frac{1}{|\mathbb{F}|} \\ &= \frac{s_0}{|\mathbb{F}|} + \sum_{i=0}^{d-1} 2s_{i+1} \frac{2}{|\mathbb{F}|} + \sum_{i=1}^{d-1} \frac{1}{|\mathbb{F}|} \\ &= \frac{s_0}{|\mathbb{F}|} + \sum_{i=0}^{d-1} \lg(S_{i+1}) \frac{2}{|\mathbb{F}|} + \sum_{i=1}^{d-1} \frac{1}{|\mathbb{F}|} \\ &\leq \frac{\lg(S)}{|\mathbb{F}|} + \sum_{i=0}^d \frac{2 \lg(S)}{|\mathbb{F}|} + \sum_{i=1}^d \frac{1}{|\mathbb{F}|} \\ &= \frac{\lg(S)}{|\mathbb{F}|} + \frac{2d \lg(S)}{|\mathbb{F}|} + \frac{d}{|\mathbb{F}|} \\ &= \frac{3d \lg(S) + d}{|\mathbb{F}|} \end{aligned}$$

5. Productcheck - A Specialized GKR Protocol

We have already seen the power of the sumcheck protocol in the above section, but what if you wanted to prove a product rather than a sum?

$$y \stackrel{?}{=} \prod_{\mathbf{b} \in \mathbb{B}^{\lg(n)}} w(\mathbf{b})$$

We will call this argument a *productcheck*, but it's also sometimes called a *Grand Product Argument* in the literature. It was central to the Plonk[10] proof system. However, for a grand product of n values, the protocol presented in Plonk took $O(n \cdot \lg(n))$ time for the prover whereas we desire a stricter bound of $O(n)$. Incredibly, this is achieved with a sumcheck, once again highlighting the utility of the protocol.

If we consider GKR for a circuit of binary multiplication gates³, i.e. a grand product, then we can bring down the runtime of both the prover and verifier to our desired bounds. With a linear prover $O(n)$ and a sublinear verifier (except for the final round which takes $O(n)$ time) meaning that the verifier's runtime will be bounded by $O(\lg^2(n) + n)$. This was originally shown by Thaler in 2013[6].

The next couple of subsections show how to specialize GKR for this purpose and go into depth on how the prover is optimized. The primary purpose of this section is not just to show that it is possible, but concretely *how* it is possible. If you're happy to accept that such a GKR-protocol can be constructed, feel free to skip this section.

5.1. The Protocol

If we consider the circuit consisting of a binary tree of multiplication gates as in Figure 1. We can present a simpler expression than that of regular GKR:

$$\tilde{W}_i(\mathbf{a}) = \sum_{\mathbf{b} \in \mathbb{B}^i} \tilde{e}\mathbf{q}(\mathbf{a}, \mathbf{b}) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 0) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 1)$$

It's clear that this expression is equivalent to the one from regular GKR, but only for the binary tree of multiplication gates. Using Lemma 4.1.1^o we get the following polynomial:

$$\begin{aligned} q(\mathbf{r}_{i-1}) &= \tilde{W}_{i+1}(\mathbf{r}_{i-1} \parallel 0) + \alpha \tilde{W}_{i+1}(\mathbf{r}_{i-1} \parallel 1) \\ &= \left(\sum_{\mathbf{b} \in \mathbb{B}^i} \tilde{e}\mathbf{q}_{(\mathbf{r}_{i-1} \parallel 0)}(\mathbf{b}) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 0) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 1) \right) + \\ &\quad \left(\sum_{\mathbf{b} \in \mathbb{B}^i} \alpha \cdot \tilde{e}\mathbf{q}_{(\mathbf{r}_{i-1} \parallel 1)}(\mathbf{b}) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 0) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 1) \right) \\ &= \sum_{\mathbf{b} \in \mathbb{B}^i} \left(\tilde{e}\mathbf{q}_{(\mathbf{r}_{i-1} \parallel 0)}(\mathbf{b}) + \alpha \cdot \tilde{e}\mathbf{q}_{(\mathbf{r}_{i-1} \parallel 1)}(\mathbf{b}) \right) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 0) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 1) \end{aligned}$$

Which means that our sumcheck polynomial is:

$$f_i(\mathbf{b}) = \left(\tilde{e}\mathbf{q}_{(\mathbf{r}_{i-1} \parallel 0)}(\mathbf{b}) + \alpha \cdot \tilde{e}\mathbf{q}_{(\mathbf{r}_{i-1} \parallel 1)}(\mathbf{b}) \right) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 0) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 1)$$

But what about round zero? Well, since round zero is always only a single gate, the prover/verifier pair doesn't need to perform a sumcheck at all. The prover can simply send $m_0 \in \mathbb{F}$, the output of the circuit, along with the evaluations of $\tilde{W}_1(0), \tilde{W}_1(1)$. Since checking these evaluations recursively is already round one's job, we're still good:

³You might reasonably wonder why not just replace the addition operation in sumcheck with a multiplication operation? The answer is that the degree of each g_1, \dots, g_ℓ polynomial would blow up, destroying the succinctness of the argument.

$$\begin{aligned}
\tilde{W}_0(x) &= \sum_{b \in \mathbb{B}^0} \tilde{e}_q(x, b) \cdot \tilde{W}_1(b \parallel 0) \cdot \tilde{W}_1(b \parallel 1) \\
&= \tilde{e}_q(x, ()) \cdot \tilde{W}_1(() \parallel 0) \cdot \tilde{W}_1(() \parallel 1) \\
&= \tilde{W}_1(0) \cdot \tilde{W}_1(1)
\end{aligned}$$

We skip discussions of soundness and completeness as there's nothing notably different from what was argued in Section 4.4, but the full protocol is still outlined below:

Prover(\hat{x}, w)

$\hat{x} := (d)$

Verifier(\hat{x}, w)

===== **Preprocessing** =====

The prover sends evaluations $v_0^{(0)}, v_1^{(0)}$ claiming that $v_0^{(0)} = \tilde{W}_1(0), v_1^{(0)} = \tilde{W}_1(1)$ and thus $y = v_0^{(0)} \cdot v_1^{(0)}$.

$$v_0^{(0)} := \tilde{W}_1(0), v_1^{(0)} := \tilde{W}_1(1) \xrightarrow{v_0^{(0)}, v_1^{(0)}} m_0 := v_0^{(0)} \cdot v_1^{(0)}$$

===== **Round 1** =====

To verify the values $v_0^{(0)}, v_1^{(0)}$ from the previous round, \mathcal{V} picks out a randomly sampled value α and sends it to \mathcal{P} . \mathcal{P} uses α to construct the combined-claim polynomial, for which \mathcal{P} and \mathcal{V} perform the sumcheck over:

$$f_1(\mathbf{b}) = (\tilde{e}q_0(\mathbf{b}) + \alpha \cdot \tilde{e}q_1(\mathbf{b})) \cdot \tilde{W}_2(\mathbf{b} \parallel 0) \cdot \tilde{W}_2(\mathbf{b} \parallel 1)$$
$$f_1(\mathbf{b}) \xleftarrow{\alpha} \alpha \in_R \mathbb{F}$$
$$\xleftarrow{\sum_{\mathbf{b} \in \mathbb{B}^1} f_1(\mathbf{b}) \stackrel{?}{=} m_1}$$

At the end of the protocol, \mathcal{P} sends \mathcal{V} the evaluations of $\tilde{W}_2(\mathbf{r}_1 \parallel 0)$ and $\tilde{W}_2(\mathbf{r}_1 \parallel 1)$, so \mathcal{V} can make the final check in the sumcheck protocol:

$$v_0^{(1)} := \tilde{W}_2(\mathbf{r}_1 \parallel 0), v_1^{(1)} := \tilde{W}_2(\mathbf{r}_1 \parallel 1) \xrightarrow{v_0^{(1)}, v_1^{(1)}} m_i \stackrel{?}{=} \tilde{e}q_0(\mathbf{r}_1) \cdot v_0^{(1)} \cdot v_1^{(1)} + \alpha \cdot \tilde{e}q_1(\mathbf{r}_1) \cdot v_0^{(1)} \cdot v_1^{(1)}$$

===== **Round $i \in [2..d - 1]$** =====

To verify the values $v_0^{(i-1)}, v_1^{(i-1)}$ from the previous round, \mathcal{V} picks out a randomly sampled value α and sends it to \mathcal{P} . \mathcal{P} uses α to construct the combined-claim polynomial, for which \mathcal{P} and \mathcal{V} perform the sumcheck over:

$$f_i(\mathbf{b}) = (\tilde{e}q_{(\mathbf{r}_{i-1} \parallel 0)}(\mathbf{b}) + \alpha \cdot \tilde{e}q_{(\mathbf{r}_{i-1} \parallel 1)}(\mathbf{b})) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 0) \cdot \tilde{W}_{i+1}(\mathbf{b} \parallel 1)$$
$$f_i(\mathbf{b}) \xleftarrow{\alpha} \alpha \in_R \mathbb{F}$$
$$\xleftarrow{\sum_{\mathbf{b} \in \mathbb{B}^i} f_i(\mathbf{b}) \stackrel{?}{=} m_i}$$

At the end of the protocol, \mathcal{P} sends \mathcal{V} the evaluations of $\tilde{W}_{i+1}(\mathbf{r}_i \parallel 0)$ and $\tilde{W}_{i+1}(\mathbf{r}_i \parallel 1)$, so \mathcal{V} can make the final check in the sumcheck protocol:

$$v_0^{(i)} := \tilde{W}_{i+1}(\mathbf{r}_i \parallel 0), v_1^{(i)} := \tilde{W}_{i+1}(\mathbf{r}_i \parallel 1) \xrightarrow{v_0^{(i)}, v_1^{(i)}} m_i \stackrel{?}{=} \tilde{e}q_{(\mathbf{r}_{i-1} \parallel 0)}(\mathbf{r}_i) \cdot v_0^{(i)} \cdot v_1^{(i)} + \alpha \cdot \tilde{e}q_{(\mathbf{r}_{i-1} \parallel 1)}(\mathbf{r}_i) \cdot v_0^{(i)} \cdot v_1^{(i)}$$

===== **Round d** =====

At the input layer d , \mathcal{V} has two claims. \mathcal{V} constructs \tilde{W}_d from w and performs a final check.

$$v_0^{(d-1)} \stackrel{?}{=} \tilde{W}_d(\mathbf{r}_{d-1} \parallel 0) \wedge v_1^{(d-1)} \stackrel{?}{=} \tilde{W}_d(\mathbf{r}_{d-1} \parallel 1)$$

5.2. Efficiency

We reuse the same trick as in corollary 3.1.1^o, by computing tables for each term and product in the sumcheck polynomials and using them to evaluate f_i in round j of the sumcheck protocol. Crucially, we need to construct the tables required to evaluate f_i in round j of the sumcheck protocol in $O(2^{i-j})$ time.

5.2.1. Computing $\tilde{e}q$ in linear time

For round i of the GKR protocol, in round j in the sumcheck invocation, we want to evaluate f_i at $\mathbf{x} = (r_1, \dots, r_j, t, b_{j+2}, \dots, b_{s_i})$, with $t = \{0, 1, 2, 3\}$:

$$f_i(\mathbf{x}) = (\tilde{e}q_{(r_{i-1} \parallel 0)}(\mathbf{x}) + \alpha \cdot \tilde{e}q_{(r_{i-1} \parallel 1)}(\mathbf{x})) \cdot \tilde{W}_{i+1}(\mathbf{x} \parallel 0) \cdot \tilde{W}_{i+1}(\mathbf{x} \parallel 1)$$

Let $\mathbf{v} = r_{i-1} \parallel 0$ for $\tilde{e}q_{(r_{i-1} \parallel 0)}$, $\mathbf{v} = r_{i-1} \parallel 1$ for $\tilde{e}q_{(r_{i-1} \parallel 1)}$ and $\ell := |\mathbf{v}| = i$. Then the definition of $\tilde{e}q_{\mathbf{v}}$ is:

$$\tilde{e}q_{\mathbf{v}}(\mathbf{b}) = \prod_{k=1}^j (v_k r_k + (1 - v_k)(1 - r_k)) \cdot \prod_{k=j+1}^{\ell} (v_k b_k + (1 - v_k)(1 - b_k))$$

With the r_1, \dots, r_j 's set by the verifier, as required by the sumcheck protocol. Now, \mathcal{P} needs an evaluation table of $\tilde{e}q_{\mathbf{v}}((b_{j+1}, \dots, b_{\ell}))$ for each round in the sumcheck protocol. In round zero, \mathcal{P} can compute the table, $\hat{e}q^{(0)}$ by leveraging Lemma 3.1.1^o in $O(2^{\ell})$ time. Then in round j , \mathcal{P} can compute the evaluation table of $\tilde{e}q_{\mathbf{v}}^{(j)}$ using the following recurrence:

$$\hat{e}q_j[(b_{j+1}, \dots, b_{\ell})] = v_j^{-1} \cdot \hat{e}q_{j-1}[(1, b_{j+1}, \dots, b_{\ell})] \cdot (v_j r_j + (1 - v_j)(1 - r_j)) \quad (5)$$

Remark

If $v_j = 0$ then this process won't work. To circumvent this, \mathcal{V} can instead sample each v_j from \mathbb{F}^* with only negligible soundness overhead.

In $O(2^{\ell-j})$ time. To see why, we can inspect the expression $\hat{e}q_{j-1}[(1, b_{j+1}, \dots, b_{\ell})]$:

$$\begin{aligned} \hat{e}q_{j-1}[(1, b_{j+1}, \dots, b_{\ell})] &= \left(\prod_{k=1}^{j-1} (v_k r_k + (1 - v_k)(1 - r_k)) \right) \cdot \\ &\quad (v_j \cdot 1 + (1 - v_j) \cdot (1 - 1)) \cdot \\ &\quad \left(\prod_{k=j+2}^{\ell} (v_k b_k + (1 - v_k)(1 - b_k)) \right) \\ &= v_j \cdot \left(\prod_{k=1}^{j-1} (v_k r_k + (1 - v_k)(1 - r_k)) \right) \cdot \left(\prod_{k=j+2}^{\ell} (v_k b_k + (1 - v_k)(1 - b_k)) \right) \end{aligned}$$

So, substituting back into Equation 5^o, we get:

$$\hat{e}q_j[(b_{j+1}, \dots, b_{\ell})] = \left(\prod_{k=1}^j (v_k r_k + (1 - v_k)(1 - r_k)) \right) \cdot \left(\prod_{k=j+1}^{\ell} (v_k b_k + (1 - v_k)(1 - b_k)) \right)$$

As expected. This process works for both $\tilde{e}q_{(r_{i-1} \parallel 0)}$ and $\tilde{e}q_{(r_{i-1} \parallel 1)}$.

5.2.2. Computing \tilde{W} in linear time

Constructing the lookup table:

Note that in round zero of the sumcheck, the prover already has a lookup table of \widehat{W}_0 :

$$\widehat{W}_0[\mathbf{x} \in \mathbb{B}^\ell] := W_{i+1}(\mathbf{x})$$

Where $\ell = i + 1$. Define \widehat{W}_j such that:

$$\widehat{W}_j[(x_{j+1}, \dots, x_\ell)] = \widetilde{W}_{i+1}(r_1, \dots, r_j, x_{j+1}, \dots, x_\ell) \quad (6)$$

Assuming \mathcal{P} already has \widehat{W}_{j-1} , they can compute \widehat{W}_j :

$$\widehat{W}_j[(x_{j+1}, \dots, x_\ell)] := \widetilde{e}q_0(r_j) \cdot \widehat{W}_{j-1}[(0, x_{j+1}, \dots, x_\ell)] + \widetilde{e}q_1(r_j) \cdot \widehat{W}_{j-1}[(1, x_{j+1}, \dots, x_\ell)]$$

In $O(2^{\ell-j}) = O(2^{i+1-j}) = O(2^{i-j})$ time.

Using the lookup table:

Once the prover has a \widehat{W}_{j-1} lookup table, they can compute $\widetilde{W}^{(j-1)}(r_1, \dots, r_{j-1}, t, x_{j+1}, \dots, x_\ell)$:

$$\begin{aligned} \widetilde{W}(r_1, \dots, r_{j-1}, t, x_{j+1}, \dots, x_\ell) &= \sum_{\mathbf{b} \in \mathbb{B}^\ell} \widetilde{e}q(r_1, \dots, r_{j-1}, t, x_{j+1}, \dots, x_\ell, \mathbf{b}) \cdot W_{i+1}(\mathbf{b}) \\ &= \sum_{\mathbf{b} \in \mathbb{B}^\ell} \widetilde{e}q_r(b_1, \dots, b_{j-1}) \cdot \widetilde{e}q_t(b_j) \cdot \widetilde{e}q_x(b_{j+1}, \dots, b_\ell) \cdot W_{i+1}(\mathbf{b}) \\ &= \sum_{\mathbf{b} \in \mathbb{B}^\ell} \widetilde{e}q_r(b_1, \dots, b_{j-1}) \cdot \widetilde{e}q_t(b_j) \cdot W_{i+1}(b_1, \dots, b_j, x_{j+1}, \dots, x_\ell) \\ &= \sum_{\mathbf{b} \in \mathbb{B}^\ell} \widetilde{e}q_r(b_1, \dots, b_{j-1}) \cdot \widetilde{e}q_0(b_j) \cdot W_{i+1}(b_1, \dots, b_j, x_{j+1}, \dots, x_\ell) + \\ &\quad \sum_{\mathbf{b} \in \mathbb{B}^\ell} \widetilde{e}q_r(b_1, \dots, b_{j-1}) \cdot \widetilde{e}q_1(b_j) \cdot W_{i+1}(b_1, \dots, b_j, x_{j+1}, \dots, x_\ell) \\ &= \widehat{W}_{j-1}[(0, x_{j+1}, \dots, x_\ell)] \cdot \widetilde{e}q_0(t) + \widehat{W}_{j-1}[(1, x_{j+1}, \dots, x_\ell)] \cdot \widetilde{e}q_1(t) \end{aligned}$$

In constant time.

5.2.3. Putting it all together

Communication: $O(\lg^2(n))$

As argued in Section 4.3, for each round in the GKR protocol the dominating communication cost is a sumcheck with $O\left(\sum_{j=1}^i \deg_j(f_i)\right)$ communication. Since $\deg_j(f_i)$ is constant, there is $O(i)$ communication. Across all rounds we have

$$O\left(\sum_{i=1}^{d-1} i\right) = O\left(\frac{d(d+1)}{2}\right) = O(d^2) = O(\lg^2(n))$$

Prover runtime: $O(n)$

In the above subsections, we established that the prover can compute $\widehat{e}q_j$ (for both $\widetilde{e}q_{(r_{i-1} \parallel 0)}$ and $\widetilde{e}q_{(r_{i-1} \parallel 1)}$) and \widehat{W}_j in $O(2^{i-j})$ time. Then the prover can use them to evaluate:

$$f_i(\mathbf{b}) = \left(\widetilde{e}q_{(r_{i-1} \parallel 0)}(\mathbf{b}) + \alpha \cdot \widetilde{e}q_{(r_{i-1} \parallel 1)}(\mathbf{b})\right) \cdot \widetilde{W}_{i+1}(\mathbf{b} \parallel 0) \cdot \widetilde{W}_{i+1}(\mathbf{b} \parallel 1)$$

With $\mathbf{b} = (r_1, \dots, r_{j-1}, t, x_{j+1}, \dots, x_{s_i})$ in constant time. Thus, the prover spends $O(3 \cdot 2^{i-j})$ time in each round of the sumcheck protocol. For all rounds of the sumcheck protocol is $O\left(\sum_{j=1}^i 2^{i-j}\right) = O(2^{s_i} - 1) = O(2^i)$.

As argued in the sumcheck section, the dominating runtime of the prover in GKR is dedicated to the sumchecks. Since we have a binary tree of gates, $n = 2^d$. Now we can compute the prover runtime:

$$O\left(\sum_{i=0}^d 2^i\right) = O(2^{d+1}) = O(2^d) = O(n)$$

Verifier runtime: $O(\lg^2(n) + n)$

As argued in Section 4.3, the verifier workload is proportional to the communication cost $O(\lg^2(n))$, plus an evaluation of the input layer $O(n)$, plus all the evaluations of $\widetilde{\text{add}}, \widetilde{\text{mul}}$ ($O(t)$). Evaluating $\widetilde{\text{add}}, \widetilde{\text{mul}}$ is just evaluating \tilde{e}_q , which as argued in Section 3.1, can be done in $O(\ell) = O(i)$ time. This gives us:

$$O(\lg^2(n) + t + n) = O\left(\lg^2(n) + \sum_{i=1}^{d-1} i + n\right) = O(\lg^2(n) + \lg^2(n) + n) = O(\lg^2(n) + n)$$

6. Spartan

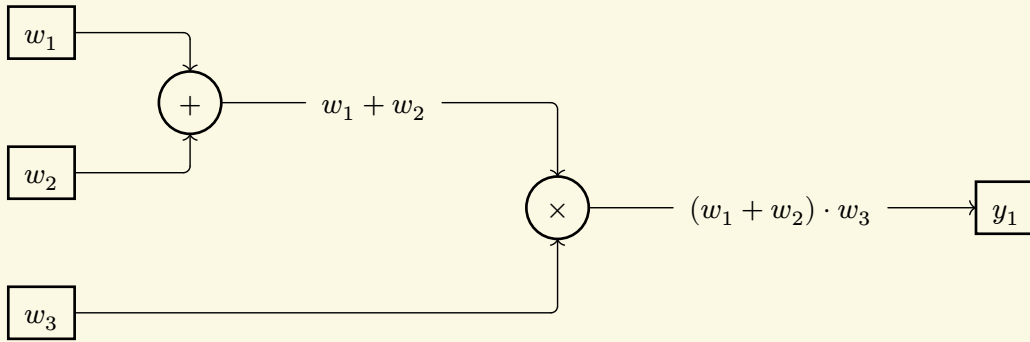
A popular alternative to GKR’s layered circuit representation is R1CS. R1CS is a constraint system that represents computation of an arithmetic circuit as a system of linear constraints combined with a single multiplication per constraint:

$$Aw \odot Bw = Cw \tag{7}$$

Where $w \in \mathbb{F}^N$ is the witness vector containing all inputs, outputs, and intermediate variables of the circuit and $A, B, C \in \mathbb{F}^{m \times N}$ are sparse matrices encoding the structure of the circuit.

Unlike GKR, which requires the circuit to be organized into layers of uniform depth, R1CS allows for “flat” structures where any variable can interact with any other. This has led to R1CS becoming a sort of *lingua franca* for SNARK circuits over the last decade or so. The approach described in this chapter specifically follows Spartan[2], which encodes R1CS as a multilinear polynomial identity and uses sumcheck to verify it.

Example 6.1 (A Small R1CS Circuit) Consider the following example circuit, which computes $y_1 = (w_1 + w_2) \cdot w_3$:



First, we define the witness vector w . It contains the constant 1, the input variables, and the output variable:

$$w = [1 \ w_1 \ w_2 \ w_3 \ y_1]^T$$

The constant one exists so that we can model constants in the circuit. Since there is only one multiplication gate in our example, the matrices will have only a single row.

- **Matrix A (Left Input):** Selects w_1, w_2 . Note, addition does not grow the dimension of the matrices.
- **Matrix B (Right Input):** Selects w_3 .
- **Matrix C (Output):** Selects y_1 .

$$A = [0 \ 1 \ 1 \ 0 \ 0], \quad B = [0 \ 0 \ 0 \ 1 \ 0], \quad C = [0 \ 0 \ 0 \ 0 \ 1]$$

To verify, we take the dot product of the row with the witness:

$$\begin{aligned}
 Cw = Aw \odot Bw &\implies \\
 \underbrace{(0 + 0 + 0 + 0 + 1 \cdot y_1)}_{\text{Output: } y_1} &= \underbrace{(0 \cdot 1 + 1 \cdot w_1 + 1 \cdot w_2 + 0 + 0)}_{\text{Left Input: } (w_1 + w_2)} \cdot \underbrace{(0 + 0 + 0 + 1 \cdot w_3 + 0)}_{\text{Right Input: } w_3} \implies \\
 y_1 &= (w_1 + w_2) \cdot w_3
 \end{aligned}$$

So to check whether the circuit is satisfied, the verifier can boil the check down to just $Cw = Aw \odot Bw$.

Our goal for the rest of this chapter is to reduce the R1CS satisfiability check to polynomial evaluations that can be verified via sumcheck. Specifically, we will encode the constraint system as a polynomial identity and ultimately use sumcheck to convince the verifier that the polynomial identity holds.

6.1. Arithmetizing R1CS

Without loss of generality, we simplify the domain of $\mathbf{A}, \mathbf{B}, \mathbf{C}$ to $\mathbb{F}^{m \times m}$ and $\mathbf{w} \in \mathbb{F}^m$, then define $s := \lg(m)$. We let n denote the number of nonzero entries in each matrix $\mathbf{A}, \mathbf{B}, \mathbf{C}$. To avoid writing everything thrice we denote $\mathbf{M} \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$. We also define $\text{toBits}(x) : \mathbb{N} \rightarrow \mathbb{B}^{\lceil \lg(x) \rceil}$ and $\text{toInt}(x) : \mathbb{B}^{|x|} \rightarrow \mathbb{N}$ representing the bijective functions between the natural numbers and their corresponding bitstrings. We can naturally express \mathbf{M}, \mathbf{w} as functions:

$$\begin{aligned} \forall \mathbf{x}, \mathbf{y} \in \mathbb{B}^s : M(\mathbf{x}, \mathbf{y}) &= M_{\text{toInt}(\mathbf{x}), \text{toInt}(\mathbf{y})} \\ \forall \mathbf{x} \in \mathbb{B}^s : w(\mathbf{x}) &= w_{\text{toInt}(\mathbf{x})} \end{aligned}$$

We now define a helpful function $F : \mathbb{B}^s \rightarrow \mathbb{F}$ which can model whether an R1CS instance is satisfied:

$$F(\mathbf{x}) = \left(\sum_{\mathbf{b} \in \mathbb{B}^s} A(\mathbf{x}, \mathbf{b}) \cdot w(\mathbf{b}) \right) \cdot \left(\sum_{\mathbf{b} \in \mathbb{B}^s} B(\mathbf{x}, \mathbf{b}) \cdot w(\mathbf{b}) \right) - \sum_{\mathbf{b} \in \mathbb{B}^s} C(\mathbf{x}, \mathbf{b}) \cdot w(\mathbf{b})$$

The R1CS instance is satisfied if and only if

$$\forall \mathbf{x} \in \mathbb{B}^s : F(\mathbf{x}) = 0 \quad (8)$$

To see why, recall that $F(\mathbf{x})$ computes the difference between the left-hand and right-hand sides of the R1CS constraint at row $\text{toInt}(\mathbf{x})$:

$$F(\mathbf{x}) = (\mathbf{A}\mathbf{w})_{\text{toInt}(\mathbf{x})} \cdot (\mathbf{B}\mathbf{w})_{\text{toInt}(\mathbf{x})} - (\mathbf{C}\mathbf{w})_{\text{toInt}(\mathbf{x})}$$

So $F(\mathbf{x}) = 0$ for all $\mathbf{x} \in \mathbb{B}^s$ simply asserts that every row of Equation 7° holds, i.e. $\mathbf{C}\mathbf{w} = \mathbf{A}\mathbf{w} \odot \mathbf{B}\mathbf{w}$.

We can of course also define the multilinear extensions of $A, B, C : \mathbb{B}^s \times \mathbb{B}^s \rightarrow \mathbb{F}, w : \mathbb{B}^s \rightarrow \mathbb{F}$ and model F as a polynomial:

$$\begin{aligned} \tilde{M}(\mathbf{x}, \mathbf{y}) &= \sum_{\mathbf{a}, \mathbf{b} \in \mathbb{B}^s} M(\mathbf{a}, \mathbf{b}) \cdot \tilde{\text{eq}}(\mathbf{x}, \mathbf{a}) \cdot \tilde{\text{eq}}(\mathbf{y}, \mathbf{b}) \\ \tilde{w}(\mathbf{x}) &= \sum_{\mathbf{b} \in \mathbb{B}^s} w(\mathbf{b}) \cdot \tilde{\text{eq}}(\mathbf{x}, \mathbf{b}) \\ f(\mathbf{x}) &= \left(\sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{A}(\mathbf{x}, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) \right) \cdot \left(\sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{B}(\mathbf{x}, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) \right) - \sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{C}(\mathbf{x}, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) \end{aligned}$$

If the R1CS instance is satisfied, $f(\mathbf{x}) = 0$ for all $\mathbf{x} \in \mathbb{B}^s$. It might be tempting to apply Schwartz-Zippel directly to f by checking if $f(\gamma) = 0$ for a random $\gamma \in \mathbb{F}^s$. However, because f multiplies the \tilde{A} and \tilde{B} sums, it has degree 2 in each variable of \mathbf{x} . A polynomial can evaluate to zero on the entire boolean hypercube without being the identically zero polynomial globally. Thus, $f(\gamma)$ will likely be nonzero even for valid proofs, rejecting a satisfied R1CS instance.

Instead, we need a polynomial that is the zero-polynomial *if and only if* the hypercube evaluations are zero. This is yet another useful property of multilinear polynomials. A multilinear polynomial is uniquely determined by its values on the boolean hypercube, meaning the multilinear extension of f , denoted \tilde{f} , is the zero polynomial if and only if the R1CS instance is satisfied.

We can then safely use Schwartz-Zippel on \tilde{f} by evaluating it at a random point.

$$\tilde{f}(\mathbf{x}) = \sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{\text{eq}}(\mathbf{x}, \mathbf{b}) \cdot f(\mathbf{b})$$

By running the sumcheck protocol, the prover can convince the verifier that $\tilde{f}(\gamma) = 0$ for a random challenge γ .

6.2. Defining the Sumcheck Polynomials g_1, g_2

The above exposition established that if the prover succeeds in convincing the verifier that the following equation holds:

$$\sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{\text{eq}}(\gamma, \mathbf{b}) \cdot f(\mathbf{b}) \stackrel{?}{=} 0 \quad (9)$$

Then, the verifier will now be convinced that Equation 7^o holds, i.e. the circuit is satisfied. The prover can of course apply sumcheck to Equation 9^o in order to convince the verifier of this fact. However, in the final round of the sumcheck, the verifier needs to compute the evaluation of the sumcheck polynomial (g_1) at a uniformly random point (ζ), chosen by the verifier:

$$g_1(\zeta) = \tilde{\text{eq}}(\gamma, \zeta) \cdot f(\zeta) \quad (10)$$

The verifier can of course evaluate $\tilde{\text{eq}}(\gamma, \zeta)$ on their own in time $O(\lg(m))$. As for $f(\zeta)$:

$$f(\zeta) = \left(\sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{A}(\zeta, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) \right) \cdot \left(\sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{B}(\zeta, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) \right) - \sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{C}(\zeta, \mathbf{b}) \cdot \tilde{w}(\mathbf{b})$$

Which can be simplified to:

$$f(\zeta) = \bar{A}(\zeta) \cdot \bar{B}(\zeta) - \bar{C}(\zeta)$$

Using the following helper polynomials:

$$\bar{A}(\mathbf{x}) := \sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{A}(\mathbf{x}, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}), \quad \bar{B}(\mathbf{x}) := \sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{B}(\mathbf{x}, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}), \quad \bar{C}(\mathbf{x}) := \sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{C}(\mathbf{x}, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) \quad (11)$$

Now, an evaluation of f simply boils down to evaluating these three polynomials at the same point. We will use a trick to reduce these three claims down to a single claim. But first, we'll show why this sumcheck will also have a linear-time prover.

Lemma 6.2.1 A sumcheck performed on g_1 will have a linear-time prover running in time $O(n + m)$.

Proof. For each matrix $M \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, compute the corresponding product $\mathbf{t}_M = M\mathbf{w}$. Since each matrix is sparse with a total of n nonzero entries, computing each of these products takes $O(n)$ time via sparse matrix-vector multiplication. As usual denote $\forall \mathbf{b} \in \mathbb{B}^s : t_M(\mathbf{b}) = (\mathbf{t}_M)_{\text{toInt}(\mathbf{b})}$. Now, note that for each $\bar{M} \in \{\bar{A}, \bar{B}, \bar{C}\}$:

$$\forall \mathbf{b} \in \mathbb{B}^s : \bar{M}(\mathbf{b}) = t_M(\mathbf{b}) \quad (12)$$

This means the prover can build a lookup table for each \bar{M} from the corresponding \mathbf{t}_M :

$$\tilde{t}_M(\mathbf{x}) = \sum_{\mathbf{b} \in \mathbb{B}^s} \tilde{\text{eq}}(\mathbf{x}, \mathbf{b}) \cdot t_M(\mathbf{b})$$

Since Equation 12^o holds, then it must also be true that the polynomials \tilde{t}_M and \bar{M} are equal, i.e. $\tilde{t}_M = \bar{M}$. Since we can use $\mathbf{t}_A, \mathbf{t}_B, \mathbf{t}_C$ as lookup tables, we can use techniques similar to the ones applied in

Section 5.2.1 and Section 5.2.2, to compute lookup tables for $\hat{e}q$ and each \hat{t}_M in time $O(m)$ and $O(n)$ respectively. This of course means that we can compute lookup tables for $\bar{A}, \bar{B}, \bar{C}$, evaluate g_1 in constant time and have a prover runtime of $O(n + m)$ for the sumcheck. \square

In the final round, the verifier needs to evaluate $g_1(\zeta) = \tilde{e}q(\gamma, \zeta)(v_{\bar{A}} \cdot v_{\bar{B}} - v_{\bar{C}})$, where the prover sends:

$$v_{\bar{A}} = \bar{A}(\zeta), \quad v_{\bar{B}} = \bar{B}(\zeta), \quad v_{\bar{C}} = \bar{C}(\zeta)$$

But the verifier then needs to verify that these are defined as in Equation 11^o. This can once again be boiled down to a sumcheck:

$$\begin{aligned} v_{\bar{A}} + \alpha \cdot v_{\bar{B}} + \alpha^2 \cdot v_{\bar{C}} &\stackrel{?}{=} \sum_{\mathbf{b} \in \mathbb{B}^s} \bar{A}(\zeta, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) + \alpha \cdot \bar{B}(\zeta, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) + \alpha^2 \cdot \bar{C}(\zeta, \mathbf{b}) \cdot \tilde{w}(\mathbf{b}) \\ &\stackrel{?}{=} \sum_{\mathbf{b} \in \mathbb{B}^s} (\bar{A}(\zeta, \mathbf{b}) + \alpha \cdot \bar{B}(\zeta, \mathbf{b}) + \alpha^2 \cdot \bar{C}(\zeta, \mathbf{b})) \cdot \tilde{w}(\mathbf{b}) \end{aligned} \quad (13)$$

By utilizing a lemma which is quite similar to Lemma 4.1.1^o:

Lemma 6.2.2 (Multiple Polynomials Evaluated at the Same Point) For polynomials p_1, p_2, \dots, p_k , each of ℓ variables, if a prover wants to convince a verifier of k claims $v_1 = p_1(\mathbf{r}), v_2 = p_2(\mathbf{r}), \dots, v_k = p_k(\mathbf{r})$ (all evaluated at the same point \mathbf{r}), then they can reduce this to a single claim over a polynomial, using a randomly sampled $\alpha \in_R \mathbb{F}$:

$$q(\mathbf{x}) := p_1(\mathbf{x}) + \alpha \cdot p_2(\mathbf{x}) + \alpha^2 \cdot p_3(\mathbf{x}) + \dots + \alpha^{k-1} \cdot p_k(\mathbf{x})$$

\mathcal{V} can then check that $q(\mathbf{r}) = v_1 + \alpha \cdot v_2 + \alpha^2 \cdot v_3 + \dots + \alpha^{k-1} \cdot v_k$. The claim that $v_i = p_i(\mathbf{r})$ for all i will then hold except with negligible probability of $\frac{k-1}{|\mathbb{F}|}$, given that q is defined as above.

Proof. If $q(\mathbf{r}) = v_1 + \alpha \cdot v_2 + \dots + \alpha^{k-1} \cdot v_k$ but the claim does not hold for some i , i.e. $v_i \neq p_i(\mathbf{r})$, then the following univariate polynomial in α is nonzero:

$$e(X) = (v_1 + X \cdot v_2 + \dots + X^{k-1} \cdot v_k) - (p_1(\mathbf{r}) + X \cdot p_2(\mathbf{r}) + \dots + X^{k-1} \cdot p_k(\mathbf{r}))$$

Since $e(X)$ has degree at most $k - 1$, by the Schwartz-Zippel lemma:

$$\Pr[e(\alpha) = 0 \mid e(X) \neq 0] \leq \frac{k-1}{|\mathbb{F}|}$$

\square

Applying sumcheck to Equation 13^o yields our second sumcheck polynomial:

$$g_2(\mathbf{x}) = (\bar{A}(\zeta, \mathbf{x}) + \alpha \cdot \bar{B}(\zeta, \mathbf{x}) + \alpha^2 \cdot \bar{C}(\zeta, \mathbf{x})) \cdot \tilde{w}(\mathbf{x}) \quad (14)$$

Lemma 6.2.3 A sumcheck performed on g_2 will have a linear-time prover running in time $O(n + m)$.

Proof. To evaluate the sumcheck for $g_2(\mathbf{x})$ efficiently, the prover must construct the evaluation form (lookup table) of the matrix polynomials inside the parenthesis ($\tilde{M}(\zeta, \mathbf{x})$).

Naively computing the lookup table \hat{M}_ζ by iterating over all 2^s entries of the domain \mathbb{B}^s for ζ and \mathbf{x} would take $O((2^s)^2)$ time. However, we can exploit the sparsity of the matrices. Let M_{nz} be the set of nonzero entries for a matrix M :

$$M_{nz} = \{(\text{val}_1, \text{row}_1, \text{col}_1), \dots, (\text{val}_n, \text{row}_n, \text{col}_n)\}$$

Where the tuple entries correspond to value, row index and column index of the given nonzero entry. The multilinear extension of a matrix M with respect to a fixed ζ can be rewritten as a sum over these sparse entries:

$$\begin{aligned} \tilde{M}(\zeta, \mathbf{x}) &= \sum_{\mathbf{a}, \mathbf{b} \in \mathbb{B}^s} M(\mathbf{a}, \mathbf{b}) \cdot \tilde{\text{eq}}(\zeta, \mathbf{a}) \cdot \tilde{\text{eq}}(\mathbf{x}, \mathbf{b}) \\ &= \sum_{i \in [1..n]} \text{val}_i \cdot \tilde{\text{eq}}(\zeta, \text{toBits}(\text{row}_i)) \cdot \tilde{\text{eq}}(\mathbf{x}, \text{toBits}(\text{col}_i)) \end{aligned}$$

This holds because the nonzero entries capture all contributions to the sum over the boolean hypercube. Then, using this sparse representation, we can create the lookup table \widehat{M}_ζ in $O(n + m)$ time using the algorithm below:

- 1: $O(m)$: Initialize an array \mathbf{t} of size m with zeros.
- 2: $O(n)$: $\forall i \in [1..n] : \mathbf{t}[\text{col}_i] \leftarrow \mathbf{t}[\text{col}_i] + \text{val}_i \cdot \tilde{\text{eq}}_\zeta[\text{toBits}(\text{row}_i)]$.
- 3: The resulting array \mathbf{t} is exactly \widehat{M}_ζ .

By applying this algorithm to matrices A, B , and C , the prover computes $\hat{A}, \hat{B}, \hat{C}$ in time $O(n + m)$ each. Since we have lookup tables for all terms in the sumcheck, we can once again apply the techniques from Section 5 to get a linear-time prover ($O(n + m)$). \square

While the prover can compute the sumcheck efficiently, a problem arises in the final check. At the end of the sumcheck protocol for g_2 , the verifier must evaluate the sumcheck polynomial at a random point η :

$$g_2(\eta) = (\tilde{A}(\zeta, \eta) + \alpha \cdot \tilde{B}(\zeta, \eta) + \alpha^2 \cdot \tilde{C}(\zeta, \eta)) \cdot \tilde{w}(\eta)$$

Note that the verifier also needs $\tilde{w}(\eta)$. The evaluation $\tilde{w}(\eta)$ can be handled by a polynomial commitment scheme as briefly discussed in Section 2.4.

We reduced R1CS satisfiability to two rounds of sumcheck, both with linear-time provers. The first sumcheck reduces the R1CS check to evaluations of $\bar{A}, \bar{B}, \bar{C}$, and the second reduces those to evaluations of the sparse matrix polynomials $\tilde{A}, \tilde{B}, \tilde{C}$ and the witness polynomial \tilde{w} . For the verifier to evaluate these directly, they would need to perform at least $O(n)$ work, destroying the succinctness of the verifier. To resolve this, we need a mechanism that allows the prover to commit to the nonzero entries of the matrices and prove the evaluation of sparse polynomials without the verifier iterating over the entries. This mechanism is provided by *Spark*.

7. Spark

We left off in the last section with an argument for R1CS based on sumcheck with a linear prover. This is indeed Spartan, but we're still missing the core contribution of Spartan, namely *Spark*. Spark solves our last problem, that the verifier still needs the evaluations of $\tilde{A}, \tilde{B}, \tilde{C}$ in the last round of the g_2 sumcheck. Of course, the verifier could do this themselves, but this would take at least $O(n + m)$ time, making the verifier linear. Another option is to use a PCS as described in Section 2.4, but any standard scheme would slow down the prover to at least $O(m^2)$ when doing an opening proof, since each of the matrix-polynomials is defined over $m \times m$ entries.

In Spark the prover only suffers a penalty of $O(n + m)$, meaning we get the desired prover time. Note that committing to the matrices $\tilde{A}, \tilde{B}, \tilde{C}$ is a *preprocessing* step (Section 6 of the Spartan paper), while Spark itself (Section 7 of the Spartan paper) is the *runtime* sparse polynomial commitment scheme that enables efficient evaluation at a queried point. To see Spark's contribution, let's start by looking at the multilinear extension of M :

$$\tilde{M}(\zeta, \eta) = \sum_{\mathbf{a}, \mathbf{b} \in \mathbb{B}^s} M(\mathbf{a}, \mathbf{b}) \cdot \tilde{\text{eq}}(\zeta, \mathbf{a}) \cdot \tilde{\text{eq}}(\eta, \mathbf{b})$$

This obviously cannot be represented as a sumcheck instance. We would need each factor of each term in the sum to be a polynomial, and the multilinear extension of M is \tilde{M} itself. But, we can represent the evaluation of \tilde{M} in its sparse form.

$$M_{\text{nz}} = \{(\text{val}_1, \text{row}_1, \text{col}_1), \dots, (\text{val}_n, \text{row}_n, \text{col}_n)\}$$

And model the evaluation of \tilde{M} with this form:

$$\begin{aligned} \tilde{M}(\zeta, \eta) &= \sum_{\mathbf{k} \in \mathbb{B}^{\lceil \lg(n) \rceil}} \text{val}(\mathbf{k}) \cdot e_{\text{row}}(\mathbf{k}) \cdot e_{\text{col}}(\mathbf{k}) \\ &= \sum_{\mathbf{k} \in \mathbb{B}^{\lceil \lg(n) \rceil}} \text{val}(\mathbf{k}) \cdot \tilde{\text{eq}}(\zeta, \text{row}(\mathbf{k})) \cdot \tilde{\text{eq}}(\eta, \text{col}(\mathbf{k})) \end{aligned} \tag{15}$$

Where for all $i \in [0, n - 1]$:

- $\text{val}(\text{toBits}(i)) : \mathbb{B}^{\lceil \lg(n) \rceil} \rightarrow \mathbb{F}$ maps a bitstring to the value of the i 'th nonzero entry of M .
- $\text{row}(\text{toBits}(i)) : \mathbb{B}^{\lceil \lg(n) \rceil} \rightarrow \mathbb{B}^s$ maps a bitstring to the row index of the i 'th nonzero entry of M .
- $\text{col}(\text{toBits}(i)) : \mathbb{B}^{\lceil \lg(n) \rceil} \rightarrow \mathbb{B}^s$ maps a bitstring to the column index of the i 'th nonzero entry of M .

Example 7.1 (Sparse Representation of a Small Matrix) Consider the following small matrix:

$$\mathbf{A} = \begin{bmatrix} 0 & 7 & 0 & 6 \\ 5 & 0 & 0 & 4 \\ 0 & 3 & 2 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In its sparse form (with simplified function domains of $\mathbb{F} \rightarrow \mathbb{F}$):

$$\begin{aligned} A_{\text{nz}} &= \{(7, 1, 2), (6, 1, 4), (5, 2, 1), (4, 2, 4), (3, 3, 2), (2, 3, 3), (1, 4, 3)\}, \\ \text{val} &= \{1 \rightarrow 7, 2 \rightarrow 6, 3 \rightarrow 5, 4 \rightarrow 4, 5 \rightarrow 3, 6 \rightarrow 2, 7 \rightarrow 1\}, \\ \text{row} &= \{1 \rightarrow 1, 2 \rightarrow 1, 3 \rightarrow 2, 4 \rightarrow 2, 5 \rightarrow 3, 6 \rightarrow 3, 7 \rightarrow 4\}, \\ \text{col} &= \{1 \rightarrow 2, 2 \rightarrow 4, 3 \rightarrow 1, 4 \rightarrow 4, 5 \rightarrow 2, 6 \rightarrow 3, 7 \rightarrow 3\}, \end{aligned}$$

In the preprocessing of the R1CS instance, a trusted party (sometimes the verifier itself) computes a succinct representation of the R1CS instance. This is a one-time cost that amortizes across all subsequent proofs for the same circuit and is how SNARKs are even able to achieve sublinear verification. A polynomial commitment to val can be computed at this stage, but notice that the other two products of each term depend on the challenges ζ and η .

If the prover could access a trusted RAM, consisting of all m values of

$$\begin{aligned} &[\tilde{\text{eq}}(\zeta, \text{row}(\text{toBits}(0))), \dots, \tilde{\text{eq}}(\zeta, \text{row}(\text{toBits}(m-1)))] \\ &[\tilde{\text{eq}}(\eta, \text{col}(\text{toBits}(0))), \dots, \tilde{\text{eq}}(\eta, \text{col}(\text{toBits}(m-1)))] \end{aligned}$$

Then we *could* perform sumcheck of the sum in Equation 15°. The next section will introduce the concept of *offline memory checking*, which solves this problem and lies at the heart of Spark. Offline memory checking also serves as the backbone of Jolt, since when we need to model instruction sets, we also need to handle reads and writes to registers. Jolt is not explicitly discussed in this document, but it largely builds upon offline memory checking and Lasso, the latter of which is covered in Section 8.

7.1. Offline Memory Checking

Offline memory checking [7] allows a potentially malicious prover (\mathcal{P}) to control a RAM for the verifier to access. The verifier (\mathcal{V}) can read and write to this RAM and verify that each read accesses the last write performed on that address. We can model a RAM as a list of values:

$$\mathbf{RAM} = [v_1, \dots, v_m]$$

We can include *timestamps* in this list corresponding to the time that the last write occurred at an address. The timestamp is a global monotonically increasing counter, where each read or write operation increments it:

$$\mathbf{RAM} = [(v_1, t_1), \dots, (v_m, t_m)]$$

Of course, it's equally valid to model this as a multi-set with an extra tuple value indicating the address:

$$\mathbf{RAM} = \{(a_1, v_1, t_1), \dots, (a_m, v_m, t_m)\} = \{(1, v_1, t_1), \dots, (m, v_m, t_m)\}$$

This is what the untrusted RAM stores controlled by \mathcal{P} , \mathcal{V} can then use the following algorithms to modify this untrusted RAM by performing reads or writes:

READ(a)

- 1: $\mathcal{P} \rightarrow \mathcal{V}$: The verifier receives value v_{read} and timestamp t from the prover.
 - 2: \mathcal{V} : The verifier adds the tuple (a, v_{read}, t) to its local set RS.
 - 3: \mathcal{V} : The verifier updates its local timestamp counter, i.e. $ts_{\text{new}} \leftarrow \max(ts, t) + 1$.
 - 4: \mathcal{V} : The verifier adds the new tuple $(a, v_{\text{read}}, ts_{\text{new}})$ to its local set WS.
 - 5: $\mathcal{V} \rightarrow \mathcal{P}$: The verifier sends $(a, v_{\text{read}}, ts_{\text{new}})$ back to the prover.
-

WRITE(a, v)

- 1: $\mathcal{P} \rightarrow \mathcal{V}$: The verifier receives value v_{read} and timestamp t from the prover.
 - 2: \mathcal{V} : The verifier adds the tuple (a, v_{read}, t) to its local read set RS.
 - 3: \mathcal{V} : The verifier updates its local timestamp counter, i.e. $ts_{\text{new}} \leftarrow \max(ts, t) + 1$.
 - 4: \mathcal{V} : The verifier adds the new tuple (a, v, ts_{new}) to its local write set WS.
 - 5: $\mathcal{V} \rightarrow \mathcal{P}$: The verifier sends (a, v, ts_{new}) back to the prover.
-

Figure 3: Verifier procedure for reading and writing to the untrusted RAM.

Here, \mathcal{V} locally stores and modifies the sets WS, RS. We also denote the sets Init, Audit which represent the initial writes and a final read pass respectively, giving \mathcal{V} the following sets:

- Init = $\{(a, v_{\text{initial}}, 0)\}$ Represents the initial write of all values to the RAM.
RS = $\{(a, v_{\text{read}}, t)\}$ The Read Set; the tuples taken from RAM.
WS = $\{(a, v, ts_{\text{new}})\}$ The Write Set; the tuples put back into RAM.
Audit = $\{(a, v_{\text{final}}, t_{\text{final}})\}$ A read pass on the final state of the RAM.

After performing the desired numbers of reads and writes, \mathcal{V} can perform the following multiset equality check:

$$\text{Init} \cup \text{WS} \stackrel{?}{=} \text{RS} \cup \text{Audit} \quad (16)$$

One way to view Equation 16^o is as a coin-mint. Each entry in the sets is a “coin” and each time \mathcal{V} adds a tuple to WS a coin is “minted” and each time \mathcal{V} adds a tuple to RS a coin is “spent”.

Init: Coins minted initially	RS: Spent coins
WS: Minted coins throughout	Audit: Unspent coins

By this logic, Equation 16^o checks that, in total, each coin spent (memory read) is exactly equal to each coin minted (memory written):

$$\text{Coins Minted Initially} + \text{Coins Minted Throughout} \stackrel{?}{=} \text{Coins Spent} + \text{Coins Unspent}$$

And intuitively fake coins would not have a corresponding member in the “Coins Minted” sets. More formally, we state that following the protocol in Equation 16^o ensures *read-consistency* with probability one.

Definition 7.1.1 (Read Consistency) Any value read from the RAM will always be the most recently written value.

Theorem 7.1.2 A verifier interacting with a potentially malicious prover that manages the RAM by leveraging the protocol above will have read-consistency with probability one.

Proof. Assume that for some read, \mathcal{P} was able to convince \mathcal{V} that a wrong value was correctly read, i.e. the value at address a , was v' when it was actually v .

There are only two cases in which the prover could cheat. They can either fabricate fake values that were never in the RAM to begin with, or they can try to convince the verifier a current value is a previous (but at the time, valid) value. These are the only two cases as we assume that for any value, it must either have never been in the RAM (fake value) or it was previously valid.

Fake value case: \mathcal{P} sends (v_{fake}, t) in step one of the Read protocol. The verifier then adds (a, v_{fake}, t) to the RS set. But this value was never written to the set WS, so the check in Equation 16° will fail with probability one.

Previously valid value case: \mathcal{P} sends $(v_{\text{old}}, t_{\text{old}})$ in step one of the Read protocol. The verifier then adds $(a, v_{\text{old}}, t_{\text{old}})$ to the RS set. This means that RS has two entries of $(a, v_{\text{old}}, t_{\text{old}})$, but since timestamps are always increasing, WS will only have a single entry. Thus the check in Equation 16° will fail with probability one. \square

The sets WS, RS quickly grow to be potentially bigger than the size of the RAM, which might make you wonder why this approach is viable at all. The answer is that we can store these sets as a digest instead. Each time we add to each set we append the element to the running hash and in the end of the protocol we compare the digests rather than the multi-sets.

7.2. Constructing a Sparse Polynomial Commitment Scheme

In Spark, we apply the offline memory-checking primitive in an alternative way. Here, the prover itself is the one that reads a public read-only RAM, and the prover wishes to convince the verifier that these reads were performed correctly. Specifically, the algorithms in Figure 3 are run by a trusted party and the resulting data is committed. Then the prover wishes to prove to a verifier that Equation 16° passes.

To do this, we must first be able to prove the equality of multisets as an argument. Furthermore, upon further inspection, you might notice that each entry in the multisets are tuples, so we also need an argument for proving tuple equality. The two lemmas below handle each of these cases:

Lemma 7.2.1 If a prover holds two n -length tuples and wishes to prove that they're equal:

$$\begin{aligned} \mathbf{a} &= (a_1, \dots, a_n) \\ \mathbf{b} &= (b_1, \dots, b_n) \end{aligned}$$

That is $\forall i \in [1..n] : a_i = b_i$, the prover can try to convince the verifier that:

$$\sum_{i=1}^n \alpha^{i-1} \cdot a_i \stackrel{?}{=} \sum_{i=1}^n \alpha^{i-1} \cdot b_i \tag{17}$$

For a uniformly random α and with soundness bound $n - 1/|\mathbb{F}|$ and completeness of one.

Proof. Completeness is trivial since two sums of equal values are equal. As for soundness: notice that each side in Equation 17° is a univariate polynomial evaluated at α and with coefficients of \mathbf{a}, \mathbf{b} respectively. If the polynomials are equal, then the coefficients, which represent each element in the list, are equal. By Schwartz-Zippel, the probability of the check passing, given that the claim does not hold is $n - 1/|\mathbb{F}|$. \square

Lemma 7.2.2 If a prover holds two multisets and wishes to prove that they're equal:

$$F = \{a_1, \dots, a_n\}$$

$$G = \{b_1, \dots, b_n\}$$

That is that there exists some permutation, π , s.t. $\forall i \in [1..n] : F_i = G_{\pi(i)}$. Let \tilde{f}, \tilde{g} be the multilinear extensions of F, G respectively. The prover can try to convince the verifier that:

$$\prod_{\mathbf{b} \in \mathbb{B}^{[\lg(n)]}} \tilde{f}(\mathbf{b}) - \beta \stackrel{?}{=} \prod_{\mathbf{b} \in \mathbb{B}^{[\lg(n)]}} \tilde{g}(\mathbf{b}) - \beta$$

For a uniformly random β and with soundness bound $n - 1/|\mathbb{F}|$ and completeness of one.

Proof. Completeness is trivial since two grand products of equal values are equal. As for soundness: we can interpret each side of the equality as a polynomial variable in β :

$$p(X) = \prod_{\mathbf{b} \in \mathbb{B}^{[\lg(n)]}} \tilde{f}(\mathbf{b}) - X$$

$$q(X) = \prod_{\mathbf{b} \in \mathbb{B}^{[\lg(n)]}} \tilde{g}(\mathbf{b}) - X$$

Then by Schwartz-Zippel, if we consider $e(X) = p(X) - q(X)$ then $e(\beta) = 0$ implies that $p = q$ with probability $1 - n - 1/|\mathbb{F}|$. Now, we just need to prove that $p = q \implies \{a_1, \dots, a_n\} = \{b_1, \dots, b_n\}$.

Consider the roots of p and q , starting with p :

$$p(X) = \prod_{(b_1, \dots, b_{[\lg(n)]}) \in \mathbb{B}^{[\lg(n)]}} \tilde{f}(b_1, \dots, b_{[\lg(n)]}) - X$$

This polynomial evaluates to zero only if one of the factors equals zero, so the roots are:

$$\begin{aligned} \text{roots}(p) &= \{ \tilde{f}(0, \dots, 0), \tilde{f}(0, \dots, 1), \dots, \tilde{f}(1, \dots, 1) \} \\ &= \{ a_1, \quad a_2, \quad \dots, \quad a_n \quad \}, \\ \text{roots}(q) &= \{ \tilde{g}(0, \dots, 0), \tilde{g}(0, \dots, 1), \dots, \tilde{g}(1, \dots, 1) \} \\ &= \{ b_1, \quad b_2, \quad \dots, \quad b_n \quad \} \end{aligned}$$

Since the two polynomials are equal, they must have the same roots. Thus:

$$\begin{aligned} \text{roots}(p) = \text{roots}(q) &\implies \\ \{a_1, \dots, a_n\} &= \{b_1, \dots, b_n\} \end{aligned}$$

\square

Combining Lemma 7.2.1° and Lemma 7.2.2° a prover can prove that:

$$\text{Init} \cup \text{WS} \stackrel{?}{=} \text{RS} \cup \text{Audit}$$

With the verifier checking whether all the elements of the read set and all the elements of the write set, are equal to some claimed value h :

$$h \stackrel{?}{=} \prod_{(a,v,t) \in \text{RS} \cup \text{Audit}} (a + \alpha v + \alpha^2 t - \beta) \stackrel{?}{=} \prod_{(a,v,t) \in \text{Init} \cup \text{WS}} (a + \alpha v + \alpha^2 t - \beta)$$

Which is an excellent use-case for the productcheck protocol from Section 5, since it can efficiently prove the correctness of a grand product of field elements.

7.3. Putting the Pieces Together

The first thing to notice before we start piecing together the puzzle is that our RAM is read-only. This means that we don't need the "WRITE" algorithm from Figure 3 and the $\max(ts, t)$ will always be ts . This is useful on its own, a prover *could* prove that they used the RAM honestly by proving correct execution of each "READ", without having to prove a max operation, which is cumbersome to do in SNARKs.

In our case this is not too important, as a trusted party/the verifier is committing to M themselves, but the same line of thinking can lead us to another optimization. We can turn the timestamps into *counters*, such that each address has its own counter/timestamp.

Example 7.3.1 (Global Timestamps vs Counters) To see the difference between the global timestamp (as described in Figure 3) and counters consider the following small example RAM, consisting of (value, timestamp/counter) pairs:

(1, 0)	(7, 0)	(2, 0)	(9, 0)
--------	--------	--------	--------

Indexing from one and performing READ(1), READ(1), READ(1), READ(4) would give us:

(1, 3)	(7, 0)	(2, 0)	(9, 4)
--------	--------	--------	--------

If we use a single global timestamp. But if we use counters, then we would get:

(1, 3)	(7, 0)	(2, 0)	(9, 1)
--------	--------	--------	--------

You can safely assume that this does not affect the result we achieved in Theorem 7.1.2^o (in fact we use it to prove a stronger version in Section 8.1). This helpfully allows us to express `write_ts` as `write_ts = read_ts + 1` and thus the prover can avoid committing to it entirely. With this in mind, we can start defining our multisets. For the RAM contents, for all $i \in [0, m - 1]$:

$$\begin{aligned} \widetilde{\text{id}}(\text{toBits}(i)) &= i \\ \widetilde{\text{zero}}(\text{toBits}(i)) &= 0 \\ \widetilde{\text{mem}}_{\text{row}}(\text{toBits}(i)) &= \widetilde{\text{eq}}_{\zeta}(\text{toBits}(i)) \\ \widetilde{\text{mem}}_{\text{col}}(\text{toBits}(i)) &= \widetilde{\text{eq}}_{\eta}(\text{toBits}(i)) \end{aligned}$$

And for the sparse representation, for all $i \in [0, n - 1]$:

$$\begin{aligned} \widetilde{\text{row}}(\text{toBits}(i)) &= \text{row}_i \\ \widetilde{\text{col}}(\text{toBits}(i)) &= \text{col}_i \end{aligned}$$

And then define the multilinear extensions of Init, RS, WS and Audit, modeling these multisets using Lemma 7.2.1^o and Lemma 7.2.2^o. Note that there are two RAMs here, for the rows:

$$\begin{aligned}
\text{Init}_{\text{row}}(\mathbf{x}) &= \tilde{\text{id}}(\mathbf{x}) + \alpha \cdot \widetilde{\text{mem}}_{\text{row}}(\mathbf{x}) + \alpha^2 \cdot \widetilde{\text{zero}}(\mathbf{x}) && - \beta, \\
\text{RS}_{\text{row}}(\mathbf{x}) &= \widetilde{\text{row}}(\mathbf{x}) + \alpha \cdot e_{\text{row}}(\mathbf{x}) + \alpha^2 \cdot \widetilde{\text{read_ts}}_{\text{row}}(\mathbf{x}) && - \beta, \\
\text{WS}_{\text{row}}(\mathbf{x}) &= \widetilde{\text{row}}(\mathbf{x}) + \alpha \cdot e_{\text{row}}(\mathbf{x}) + \alpha^2 \cdot (\widetilde{\text{read_ts}}_{\text{row}}(\mathbf{x}) + 1) && - \beta, \\
\text{Audit}_{\text{row}}(\mathbf{x}) &= \tilde{\text{id}}(\mathbf{x}) + \alpha \cdot \widetilde{\text{mem}}_{\text{row}}(\mathbf{x}) + \alpha^2 \cdot \widetilde{\text{audit_ts}}_{\text{row}}(\mathbf{x}) && - \beta,
\end{aligned}$$

And the columns:

$$\begin{aligned}
\text{Init}_{\text{col}}(\mathbf{x}) &= \tilde{\text{id}}(\mathbf{x}) + \alpha \cdot \widetilde{\text{mem}}_{\text{col}}(\mathbf{x}) + \alpha^2 \cdot \widetilde{\text{zero}}(\mathbf{x}) && - \beta, \\
\text{RS}_{\text{col}}(\mathbf{x}) &= \widetilde{\text{col}}(\mathbf{x}) + \alpha \cdot e_{\text{col}}(\mathbf{x}) + \alpha^2 \cdot \widetilde{\text{read_ts}}_{\text{col}}(\mathbf{x}) && - \beta, \\
\text{WS}_{\text{col}}(\mathbf{x}) &= \widetilde{\text{col}}(\mathbf{x}) + \alpha \cdot e_{\text{col}}(\mathbf{x}) + \alpha^2 \cdot (\widetilde{\text{read_ts}}_{\text{col}}(\mathbf{x}) + 1) && - \beta, \\
\text{Audit}_{\text{col}}(\mathbf{x}) &= \tilde{\text{id}}(\mathbf{x}) + \alpha \cdot \widetilde{\text{mem}}_{\text{col}}(\mathbf{x}) + \alpha^2 \cdot \widetilde{\text{audit_ts}}_{\text{col}}(\mathbf{x}) && - \beta,
\end{aligned}$$

The polynomials $\widetilde{\text{read_ts}}_{\text{row}}$, $\widetilde{\text{audit_ts}}_{\text{row}}$, $\widetilde{\text{read_ts}}_{\text{col}}$ and $\widetilde{\text{audit_ts}}_{\text{col}}$ are derived from running the Read algorithm in Figure 3 over all n nonzero entries. Then, we can use the specialized Grand Product GKR protocol of Section 5 to verify each of the below grand products (shown for the row RAM; the column RAM is analogous):

$$\begin{aligned}
\text{eval}_{\text{WS}}^{(1)} &= \prod_{\mathbf{b} \in \mathbb{B}^{\lceil \lg(m) \rceil}} \text{Init}_{\text{row}}(\mathbf{b}), & \text{eval}_{\text{WS}}^{(2)} &= \prod_{\mathbf{b} \in \mathbb{B}^{\lceil \lg(m) \rceil}} \text{WS}_{\text{row}}(\mathbf{b}), \\
\text{eval}_{\text{RS}}^{(1)} &= \prod_{\mathbf{b} \in \mathbb{B}^{\lceil \lg(m) \rceil}} \text{Audit}_{\text{row}}(\mathbf{b}), & \text{eval}_{\text{RS}}^{(2)} &= \prod_{\mathbf{b} \in \mathbb{B}^{\lceil \lg(m) \rceil}} \text{RS}_{\text{row}}(\mathbf{b}),
\end{aligned}$$

Which the verifier can use to check whether:

$$\text{eval}_{\text{WS}}^{(1)} \cdot \text{eval}_{\text{RS}}^{(2)} \stackrel{?}{=} \text{eval}_{\text{RS}}^{(1)} \cdot \text{eval}_{\text{WS}}^{(2)}$$

This shows the correctness of the memory checking. In the final round of the grand product argument, the verifier will need to evaluate each of these polynomials at a uniformly randomly chosen \mathbf{r} . Taking $\text{Init}_{\text{row}}(\mathbf{x})$ as an example:

$$\text{Init}_{\text{row}}(\mathbf{x}) = \tilde{\text{id}}(\mathbf{x}) + \alpha \cdot \widetilde{\text{mem}}_{\text{row}}(\mathbf{x}) + \alpha^2 \cdot \widetilde{\text{zero}}(\mathbf{x}) - \beta$$

We run the specialized GKR protocol over this polynomial, and at the end of the protocol, the verifier needs to evaluate Init_{row} at a randomly sampled point $\mathbf{r} \in_{\mathcal{R}} \mathbb{F}$. All three polynomials $\tilde{\text{id}}$, $\widetilde{\text{mem}}_{\text{row}}$, $\widetilde{\text{zero}}$ can be evaluated by the verifier on their own in logarithmic time:

$$\widetilde{\text{mem}}_{\text{row}}(\mathbf{r}) = \tilde{\text{eq}}_{\mathcal{C}}(\mathbf{r}), \quad \widetilde{\text{zero}}(\mathbf{r}) = 0, \quad \tilde{\text{id}}(\mathbf{r}) = \sum_{j=1}^{\lg(m)} r_j \cdot 2^{\lg(m)-j}$$

As for $\widetilde{\text{row}}$, $\widetilde{\text{col}}$, $\widetilde{\text{read_ts}}_{\text{row}}$, $\widetilde{\text{read_ts}}_{\text{col}}$, $\widetilde{\text{audit_ts}}_{\text{row}}$ and $\widetilde{\text{audit_ts}}_{\text{col}}$, they have no structure to exploit, so the verifier needs the prover to perform evaluation proofs on them on behalf of the verifier. But, a trusted party makes the commitments, so the verifier trusts that they are the right polynomials.

The remaining e_{col} and e_{row} polynomials are committed to by the prover but the verifier need not check their structure! If the prover committed to invalid polynomials here, it would be equivalent to breaking

read-consistency, which we proved was impossible in Theorem 7.1.2⁴

7.4. The Spark Protocol

We now present the formal Spark sparse polynomial commitment scheme, summarizing all the components discussed in this chapter. We start with the following helper algorithm that the prover can utilize to get the necessary timestamp polynomials:

MEMORYINTHEHEAD(M)

- 1: **Let** m denote the dimensions of the matrix $M \in \mathbb{F}^{m \times m}$
- 2: **Let** $\text{audit_ts}_{\text{row}} = \text{vec!}[0; m - 1]$, $\text{audit_ts}_{\text{col}} = \text{vec!}[0; m - 1]$
- 3: **Let** $\text{read_ts}_{\text{row}} = []$, $\text{read_ts}_{\text{col}} = []$
- 4: **For** $(i, j) \in ([0..m], [0..m])$:
- 5: **If** $M(i, j) \neq 0$:
- 6: $\text{read_ts}_{\text{row}}.\text{push}(\text{audit_ts}_{\text{row}}[i])$
- 7: $\text{audit_ts}_{\text{row}}[i] \leftarrow \text{audit_ts}_{\text{row}}[i] + 1$
- 8: $\text{read_ts}_{\text{col}}.\text{push}(\text{audit_ts}_{\text{col}}[j])$
- 9: $\text{audit_ts}_{\text{col}}[j] \leftarrow \text{audit_ts}_{\text{col}}[j] + 1$
- 10: **Return** $(\widetilde{\text{read_ts}}_{\text{row}}, \widetilde{\text{read_ts}}_{\text{col}}, \widetilde{\text{audit_ts}}_{\text{row}}, \widetilde{\text{audit_ts}}_{\text{col}})$

Then, let $\text{PC} = (\text{PC.SETUP}, \text{PC.COMMIT}, \text{PC.OPEN}, \text{PC.CHECK})$ be any extractable polynomial commitment scheme for dense multilinear polynomials. Now we can concretely construct the SPARK sparse polynomial commitment scheme.

SPARK.SETUP($1^\lambda, m, n$)

- 1: **Return** $\text{pp} \leftarrow \text{PC.SETUP}(1^\lambda, \max(\lceil \lg(m) \rceil, \lceil \lg(n) \rceil))$

SPARK.COMMIT(pp, M)

- 1: **Let** $(\widetilde{\text{val}}, \widetilde{\text{row}}, \widetilde{\text{col}})$ denote the sparse representation of \widetilde{M} as described in text.
- 2: $C_{\text{val}} \leftarrow \text{PC.COMMIT}(\text{pp}, \widetilde{\text{val}})$
- 3: $C_{\text{row}} \leftarrow \text{PC.COMMIT}(\text{pp}, \widetilde{\text{row}})$
- 4: $C_{\text{col}} \leftarrow \text{PC.COMMIT}(\text{pp}, \widetilde{\text{col}})$
- 5: **Let** $(\widetilde{\text{read_ts}}_{\text{row}}, \widetilde{\text{read_ts}}_{\text{col}}, \widetilde{\text{audit_ts}}_{\text{row}}, \widetilde{\text{audit_ts}}_{\text{col}}) \leftarrow \text{MEMORYINTHEHEAD}(M)$
- 6: $C_{\text{read_ts_row}} \leftarrow \text{PC.COMMIT}(\text{pp}, \widetilde{\text{read_ts}}_{\text{row}})$
- 7: $C_{\text{read_ts_col}} \leftarrow \text{PC.COMMIT}(\text{pp}, \widetilde{\text{read_ts}}_{\text{col}})$
- 8: $C_{\text{audit_ts_row}} \leftarrow \text{PC.COMMIT}(\text{pp}, \widetilde{\text{audit_ts}}_{\text{row}})$
- 9: $C_{\text{audit_ts_col}} \leftarrow \text{PC.COMMIT}(\text{pp}, \widetilde{\text{audit_ts}}_{\text{col}})$
- 10: **Return** $C \leftarrow (C_{\text{val}}, C_{\text{row}}, C_{\text{col}}, C_{\text{read_ts_row}}, C_{\text{read_ts_col}}, C_{\text{audit_ts_row}}, C_{\text{audit_ts_col}})$

⁴Technically possible in our case since we prove the multi-set equality using interactive arguments, but negligible.

SPARK.OPEN(pp, \tilde{M} , C , (m, n) , \mathbf{r})

1: **Let** $(\zeta, \eta) = \mathbf{r}$, where $\zeta, \eta \in \mathbb{F}^{\lg(m)}$, $\mathbf{r} \in \mathbb{F}^{2\lg(m)}$.

2: **Let** $(\tilde{\text{val}}, \tilde{\text{row}}, \tilde{\text{col}})$ denote the sparse representation of \tilde{M} as described in text.

3: **Prover:**

4: $\mathbf{e}_{\text{row}} := [\tilde{\text{eq}}_{\zeta}(\text{row}(0)), \dots, \tilde{\text{eq}}_{\zeta}(\text{row}(n-1))]$

5: $\mathbf{e}_{\text{col}} := [\tilde{\text{eq}}_{\eta}(\text{col}(0)), \dots, \tilde{\text{eq}}_{\eta}(\text{col}(n-1))]$

6: $C_{\mathbf{e}_{\text{row}}} \leftarrow \text{PC.COMMIT}(\text{pp}, \tilde{\mathbf{e}}_{\text{row}})$.

7: $C_{\mathbf{e}_{\text{col}}} \leftarrow \text{PC.COMMIT}(\text{pp}, \tilde{\mathbf{e}}_{\text{col}})$.

8: Send $C_{\mathbf{e}_{\text{row}}}, C_{\mathbf{e}_{\text{col}}}$ to \mathcal{V}

Apply the Specialized GKR protocol as described in the previous subsection to prove the correctness of $\tilde{\mathbf{e}}_{\text{row}}, \tilde{\mathbf{e}}_{\text{col}}$. Then use sumcheck to prove:

9:

$$v = \tilde{M}(\zeta, \eta) = \sum_{\mathbf{b} \in \mathbb{B}^{\lg(n)}} \tilde{\text{val}}(\mathbf{b}) \cdot \tilde{\mathbf{e}}_{\text{row}}(\mathbf{b}) \cdot \tilde{\mathbf{e}}_{\text{col}}(\mathbf{b})$$

The Specialized GKR protocol can easily be turned noninteractive using the Fiat-Shamir heuristic. It turns a public-coin (an interactive protocol where the verifier only sends uniformly sampled challenge values) interactive proof into a non-interactive proof, by replacing all uniformly random values sent from the verifier to the prover with calls to a non-interactive random oracle. In practice, a cryptographic hash function, ρ , is used.

By using the Fiat-Shamir heuristic SPARK.OPEN would produce a proof π and naturally lead to a SPARK.CHECK method.

8. Lasso

8.1. Improved Security Analysis

In the previous section we defined SPARK, the sparse PCS, and in section Section 6 we showed how such a scheme could be used to create a linear-time SNARK prover for RICS instances. However, SPARK as presented thus far has a glaring flaw which makes it impractical as a general-purpose PCS. This is due to the required assumption that a trusted party commits to \tilde{M} . This is not an issue in Spartan, where it was already assumed that a trusted party committed to the matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$. In general it would be quite useful to get a polynomial commitment scheme that takes only $O(n)$ time to perform an evaluation proof. That is, the evaluation proof is linear in the *nonzero entries* of \mathbf{M} .

This is the first important result shown in the Lasso paper, and remarkably the authors show that Spark is already such a scheme, as it maintains its security even when the prover commits to $\widetilde{\text{val}}, \widetilde{\text{row}}, \widetilde{\text{col}}, \widetilde{\text{read_ts_row}}, \widetilde{\text{read_ts_col}}, \widetilde{\text{audit_ts_row}}, \widetilde{\text{audit_ts_col}}$ themselves. While this wasn't necessary for Spartan, it's required when we apply the same principles to Lasso.

Before getting into the proof, it's important to understand that $\widetilde{\text{val}}, \widetilde{\text{row}}, \widetilde{\text{col}}$ need not be committed "honestly". If the prover commits some other polynomials $\widetilde{\text{val}}, \widetilde{\text{row}}, \widetilde{\text{col}}$ this is then just equivalent to the prover committing to another matrix \mathbf{M}' .

So, to get the desired result that the prover can perform SPARK.COMMIT themselves, we need to show that the multi-set equality check already implies read-consistency regardless of the committed timestamps. The key to this result is the local counter trick we adopted for optimization purposes (see Example 7.3.1^o).

Theorem 8.1.1 (Read-Consistency for Read-Only Memory) Let T be a memory array, k the number of reads and N the size of the memory. Assume that $k \leq |\mathbb{F}|, N \leq |\mathbb{F}|$. If the verifier enforces the multi-set equality check from Equation 16^o using local counters and where $\text{WS} = \{(a, v, t + 1) \mid (a, v, t) \in \text{RS}\}$, then the check will enforce read-consistency with probability one.

Proof. First, let the initial state of the memory be defined as:

$$\text{Init} = \{(i, \mathbf{RAM}[i], 0)\}_{i=0}^{N-1}$$

Which is enforced by the verifier.

We'll prove this using a proof of contradiction. Assume that \mathcal{P} successfully constructs RS, WS, and Audit such that the multiset equality holds, but RS contains an invalid read. Namely, there exists some tuple $\tau = (a, v^*, t) \in \text{RS}$ where $v^* \neq \mathbf{RAM}[a]$.

To satisfy $\text{Init} \cup \text{WS} \stackrel{z}{=} \text{RS} \cup \text{Audit}$, the invalid tuple τ must have a matching counterpart in the left-hand side of the equation ($\text{Init} \cup \text{WS}$). In this event there are two cases:

1. $\tau \in \text{Init}$: By definition, all tuples in Init take the form $(i, \mathbf{RAM}[i], 0)$. Since $v^* \neq \mathbf{RAM}[a]$, τ cannot be in Init.
2. $\tau \in \text{WS}$: By the structural definition of read-only updates, every tuple in WS is derived directly from a prior read in RS, taking the form $(a_i, v_i, t_i + 1)$. If $\tau = (a, v^*, t) \in \text{WS}$, then there must necessarily exist a corresponding "parent" read tuple $\tau' = (a, v^*, t - 1)$ in RS.

In the second case, if $\tau' \in \text{RS}$, then this tuple must also exist on the left-hand side of the equation. Recursively applying this logic means that the prover would have to include the tuple $\tau_0 = (a, v^*, 0)$ in RS. Then this tuple must also exist on the left-hand side, leading to two cases:

1. $\tau_0 \in \text{WS}$: Which is impossible because all tuples in WS have strictly positive timestamps ($t > 0$).
2. $\tau_0 \in \text{Init}$: Which is impossible because $v^* \neq \mathbf{RAM}[a]$.

Thus, we get read-consistency. □

You might wonder why *Audit* is not mentioned in the above proof, but this is because it's simply not relied on to achieve read-consistency. It is however necessary to guarantee that the multi-set check from Equation 16° passes in the first place; remember that the above proof is conditioned on the multi-set check passing, but this can only happen if *Audit* is well-formed.

8.2. The Lasso Lookup Argument

Suppose we wanted to prove a lookup into a table to a verifier. One way of viewing this is with a single read of a read-only RAM. Suppose the table is of size N and that N is a power of two, we could use memory-checking techniques to prove this. But imagine that this memory was extremely large, such as 2^{128} . In this case the instantiated memory from the offline memory checking would also be 2^{128} , far too large to instantiate in an interactive argument. We can use the same trick as was employed in *Spark*. However, recall that in that case, we wanted to evaluate the multilinear extension of M . We did so using two $\tilde{e}q$ polynomials, but what would happen if we used the natural MLE of M ?

$$\tilde{M}(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{a}, \mathbf{b} \in \mathbb{B}^{\lceil \lg(m) \rceil}} M(\mathbf{a}, \mathbf{b}) \cdot \tilde{e}q(\mathbf{x} \parallel \mathbf{y}, \mathbf{a} \parallel \mathbf{b})$$

We would then of course still use the sparse representation of M to compute this evaluation:

$$\tilde{M}(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{b} \in \mathbb{B}^{\lceil \lg(n) \rceil}} \text{val}(\mathbf{b}) \cdot \tilde{e}q(\mathbf{x} \parallel \mathbf{y}, \text{row}(\mathbf{b}) \parallel \text{col}(\mathbf{b}))$$

Besides the fact that the above equation looks more complicated than the one we arrived at in the *Spark* section, it also has a major disadvantage when used with our offline memory checking technique. The consequence is that we would be reading from a memory of size m^2 , and the resulting memory-checking argument would take time $O(n + m^2)$, ruining our hopes for a sparse polynomial commitment scheme. Luckily, we were able to decompose $\tilde{e}q$ using the identity: broadly

$$\tilde{e}q(\mathbf{x} \parallel \mathbf{y}, \text{row}(\mathbf{b}) \parallel \text{col}(\mathbf{b})) = \tilde{e}q(\mathbf{x}, \text{row}(\mathbf{b})) \cdot \tilde{e}q(\mathbf{y}, \text{col}(\mathbf{b}))$$

A primary insight of *Lasso* was that this trick is broadly useful. Suppose we wanted to construct a lookup argument of bitwise-XOR, and suppose we wanted to perform this lookup on two unreasonably large values of two 64-bit values giving us a truth-table of size 2^{128} . This table would of course be way too large to ever concretely instantiate or especially commit to.

But bitwise-XOR is exactly just that, bitwise, and this means that this table too, is *decomposable*. Instead of a single lookup in a table of size 2^{128} we could do eight lookups into sub-tables of size 2^{16} and then concatenate them together.

Example 8.2.1 (XOR decomposition)

If we take a simple example, where we want to do a bitwise-XOR on two inputs of size 2^8 , meaning we have a lookup on a table of size 2^{16} . This might be small enough that it won't need decomposition, but it might be useful for demonstration purposes. We can split this table in four identical sub-tables of size 2^4 :

x_i	00	00	00	00	01	01	01	01	10	10	10	10	11	11	11	11
y_i	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
z_i	00	01	10	11	01	00	11	10	10	11	00	01	11	10	01	00

Where $x_i \oplus y_i = z_i$. We can then perform eight lookups into these smaller tables and concatenate them:

$$\begin{aligned}
01000001 \oplus 00100000 &= (\hat{\oplus})_2(01, 00) \parallel (\hat{\oplus})_2(00, 10) \parallel (\hat{\oplus})_2(00, 00) \parallel (\hat{\oplus})_2(01, 00) \\
&= 01 \parallel 10 \parallel 00 \parallel 01 \\
&= 1100001
\end{aligned}$$

So, for $\tilde{e}q$ we combined the two sub-tables with multiplication, but in this case we're combining the tables using bit concatenation. Since our tables take in bits and return a field element we can model this bit concatenation in the following way:

$$\sum_{i=1}^c v_i \cdot 2^{w \cdot (c-i)}$$

Where c is the number of chunks and w is the *window size*, the number of bits per limb. Finally v represents the result of each lookup into the sub-tables. For reference, in Example 8.2.1°, $c = 4, w = 2, v = [1, 2, 0, 1]$. In general, we can abstract away how the sub-tables are recomposed with some function g and require the following to hold:

$$\hat{T}[b] = g(\hat{T}_1[\bar{b}_1], \dots, \hat{T}_c[\bar{b}_c])$$

Where $b = \bar{b}_1 \parallel \dots \parallel \bar{b}_c$, \hat{T} is the large lookup table of size N and $\hat{T}_1, \dots, \hat{T}_c$ are the small sub-tables of size $N^{1/c}$. Following this line of thought, we could already start constructing an interactive argument. We simply perform c offline memory checks as in Section 7, then let the verifier recompose using g .

But Lasso has one more trick up its sleeve: using Spark, we can actually batch k arguments into one. In general, we can view a lookup operation as a simple matrix-vector multiplication:

$$Mt = a$$

Where t is our table of size N , a is our vector of k lookup results, and M is a $k \times N$ sparse matrix where each row has exactly one 1 corresponding to the accessed index.

Example 8.2.2 (Matrix-Vector Lookup for 1-bit XOR) Let the lookup table t be the XOR truth table and the result of our lookups a . The matrix M has a single 1 per row to select the index.

$$\underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_M \cdot \underbrace{\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}}_t = \underbrace{\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}}_a$$

We can of course use the usual methods to convert this to a polynomial form:

$$\tilde{M}(x, y) \cdot \tilde{t}(y) = \tilde{a}(x)$$

Where $\mathbf{x} \in \mathbb{B}^{\lg(k)}$, $\mathbf{y} \in \mathbb{F}^{\lg(N)}$. We now wish to establish that the above equality holds, which we can of course use Schwartz-Zippel for. This means the verifier wants to check the evaluation $\tilde{a}(\mathbf{r})$ at some random point $\mathbf{r} \in_R \mathbb{B}^{\lg(N)}$ equals:

$$\tilde{a}(\mathbf{r}) \stackrel{?}{=} \sum_{\mathbf{b} \in \mathbb{B}^{\lg(k)}} \tilde{M}(\mathbf{r}, \mathbf{b}) \cdot \tilde{t}(\mathbf{b}) \quad (18)$$

Since there is only a single entry in M which is nonzero, then the above expression is the same as:

$$\tilde{a}(\mathbf{r}) \stackrel{?}{=} \sum_{\mathbf{b} \in \mathbb{B}^{\lg(k)}} \tilde{e}\tilde{q}(\mathbf{r}, \mathbf{b}) \cdot \hat{T}[\text{nz}(\mathbf{b})] \quad (19)$$

Where nz denotes the nonzero entry of each row of M . This follows from the fact that the polynomials of Equation 18^o and Equation 19^o are both MLE and agree on all points on the boolean hypercube.

As previously established because the table \hat{T} is decomposable, we can replace $\hat{T}[\text{nz}(\mathbf{b})]$ with our recomposition function g applied to c smaller sub-table lookups. Let $\tilde{e}_i(\mathbf{x})$ be the multilinear extension of the k lookups into \hat{T}_i . By substituting in the recomposition function g , the identity collapses to:

$$\tilde{a}(\mathbf{r}) = \sum_{\mathbf{b} \in \mathbb{B}^{\lg(k)}} \tilde{e}\tilde{q}(\mathbf{r}, \mathbf{b}) \cdot g(\tilde{e}_1(\mathbf{b}), \dots, \tilde{e}_c(\mathbf{b}))$$

Once the e_i sub-lookups are verified, the verifier simply runs a standard sum-check protocol over the $\lg(k)$ -variate boolean hypercube on the above sum, with the following sumcheck polynomial:

$$f_{\text{Lasso}}(\mathbf{x}) := \tilde{e}\tilde{q}(\mathbf{r}, \mathbf{x}) \cdot g(\tilde{e}_1(\mathbf{x}), \dots, \tilde{e}_c(\mathbf{x}))$$

Assuming the verifying checks pass, this proves the correctness of the lookups into the massive table \hat{T} .

8.3. Efficiency of the Lasso Lookup Argument

Prover Costs:

Assuming linear commitment costs in the number of variables (for MLE polynomials) of $O(n)$, the prover runtime is dominated by the following operations:

Commitments:	$\tilde{n}z_1, \dots, \tilde{n}z_c$	$\tilde{e}_1, \dots, \tilde{e}_c$	$\widetilde{\text{read_ts}}_1, \dots, \widetilde{\text{read_ts}}_c$	$\widetilde{\text{audit_ts}}_1, \dots, \widetilde{\text{audit_ts}}_c$
Cost:	$O(c \cdot k)$	$O(c \cdot k)$	$O(c \cdot k)$	$O(c \cdot N^{1/c})$
EvalProofs:	$\tilde{n}z_1, \dots, \tilde{n}z_c$	$\tilde{e}_1, \dots, \tilde{e}_c$	$\widetilde{\text{read_ts}}_1, \dots, \widetilde{\text{read_ts}}_c$	$\widetilde{\text{audit_ts}}_1, \dots, \widetilde{\text{audit_ts}}_c$
Cost:	$O(c \cdot k)$	$O(c \cdot k)$	$O(c \cdot k)$	$O(c \cdot N^{1/c})$
Productchecks:	$\text{RS}_1, \dots, \text{RS}_c$	$\text{WS}_1, \dots, \text{WS}_c$	$\text{Init}_1, \dots, \text{Init}_c$	$\text{Audit}_1, \dots, \text{Audit}_c$
Cost:	$O(c \cdot k)$	$O(c \cdot k)$	$O(c \cdot N^{1/c})$	$O(c \cdot N^{1/c})$
Sumchecks:	f_{Lasso}			
Cost:	$O(c \cdot k)$			

For a total of:

$$O((3 \cdot c \cdot k + c \cdot N^{1/c}) + (3 \cdot c \cdot k + c \cdot N^{1/c}) + (2 \cdot c \cdot k + 2 \cdot c \cdot N^{1/c}) + c \cdot k)$$

We can batch sumchecks, productchecks and evaluation proofs. These batching techniques follow trivially from Schwartz-Zippel. If you have n sumchecks, all over $\lg(k)$ variables:

$$\sigma_1 \stackrel{?}{=} \sum_{\mathbf{b} \in \mathbb{B}^{\lg(k)}} f_1(\mathbf{b}) \wedge \dots \wedge \sigma_n \stackrel{?}{=} \sum_{\mathbf{b} \in \mathbb{B}^{\lg(k)}} f_n(\mathbf{b})$$

Then we can batch them using a uniformly random value $\alpha \in_R \mathbb{F}$:

$$\sum_{i=1}^n \alpha^{i-1} \cdot \sigma_i \stackrel{?}{=} \sum_{\mathbf{b} \in \mathbb{B}^{\lg(k)}} \sum_{i=1}^n \alpha^{i-1} \cdot f_i(\mathbf{b})$$

The same holds for our productchecks. Since the productchecks only apply sumcheck in the same manner as the GKR protocol we can use the sumcheck batching technique on the sumcheck polynomials of the productcheck, as long as the grand products have the same number of entries.

As for the evaluation proofs, consider the evaluations:

$$v_1 = f_1(\zeta), \dots, v_n = f_n(\zeta)$$

Proved correct using evaluation proofs, with corresponding commitments:

$$C_1 = \text{PC.COMMIT}(f_1, d), \dots, C_n = \text{PC.COMMIT}(f_n, d)$$

Then we could of course verify each on its own, with an assumed linear cost of $O(k)$ each, for a total cost of $O(n \cdot k)$:

$$\text{PC.CHECK}(C_1, d, \zeta, v_1, \pi_1) \wedge \dots \wedge \text{PC.CHECK}(C_n, d, \zeta, v_n, \pi_n)$$

But, assuming we have additively homomorphic commitments, the prover could also instead construct a batching polynomial q , using a uniformly random value α .

$$q(\mathbf{x}) = \sum_{i=1}^n \alpha^{i-1} \cdot f_i(\mathbf{x})$$

Send the evaluation and proof for $g(\zeta)$ along with the evaluations v_1, \dots, v_n . The verifier can then check whether:

$$\text{PC.CHECK}(C_q, d, \zeta, q(\zeta), \pi_q) \wedge q(\zeta) \stackrel{?}{=} \sum_{i=1}^n \alpha^{i-1} \cdot v_i$$

Which has only a $O(n)$ cost for the prover. Applying these batching techniques, the cost for the prover becomes:

$$O((3 \cdot c \cdot k + c \cdot N^{1/c}) + (k + N^{1/c}) + (k + N^{1/c}) + c \cdot k)$$

Since the first term is dominant, we can simplify this to:

$$O(c \cdot k + c \cdot N^{1/c})$$

Picking c such that $N^{1/c} < c \cdot k$ means that the prover cost primarily scales with the number of lookups, not the size of the table. Another thing to note is that the primary cost to the prover is from the commitments, due to the batching techniques applied above. This is also why the Lasso paper primarily focuses on the commitment cost in their discussion.

Verifier Cost:

Assuming that an evaluation proof takes $O(\lg(n))$ in the number of variables. Let:

$$\lambda_k = \lg^2(k) + \lg(k), \quad \lambda_N = \lg^2(N^{1/c}) + \lg(N^{1/c})$$

EvalProofs:	$\tilde{n}z_1, \dots, \tilde{n}z_c$	$\tilde{e}_1, \dots, \tilde{e}_c$	$\widetilde{\text{read_ts}}_1, \dots, \widetilde{\text{read_ts}}_c$	$\widetilde{\text{audit_ts}}_1, \dots, \widetilde{\text{audit_ts}}_c$
Cost:	$O(c \cdot \lg(k))$	$O(c \cdot \lg(k))$	$O(c \cdot \lg(k))$	$O(c \cdot \lg(N^{1/c}))$
Productchecks:	$\text{RS}_1, \dots, \text{RS}_c$	$\text{WS}_1, \dots, \text{WS}_c$	$\text{Init}_1, \dots, \text{Init}_c$	$\text{Audit}_1, \dots, \text{Audit}_c$
Cost:	$O(c \cdot \lambda_k)$	$O(c \cdot \lambda_k)$	$O(c \cdot \lambda_N)$	$O(c \cdot \lambda_N)$
Sumchecks:	f_{Lasso}			
Cost:	$O(c + \lg(k))$			

With batching this then becomes:

$$O((\lg(k) + \lg(N^{1/c})) + (\lambda_k + \lambda_N) + (c + \lg(k)))$$

Expanding λ_k and λ_N and simplifying, the verifier's work is polylogarithmic in both k and $N^{1/c}$, plus a small additive $O(c)$ term from the recomposition function g .

8.4. Soundness

The overall soundness of the Lasso lookup argument follows by a union bound over its constituent sub-protocols. We can compute this loosely using Big O notation. For simplicity denote $m = \max(k, N^{1/c})$.

- **Sumcheck:** The sumcheck over $\lg(k)$ variables contributes a soundness error of at most $O(\lg(k)/|\mathbb{F}|)$.
- **Productchecks:** Each of the $4c$ grand-product arguments (for $\text{RS}_i, \text{WS}_i, \text{Init}_i, \text{Audit}_i$ across c sub-tables) invokes the specialized GKR protocol. By the analysis of Section 4.4, each contributes an error of $O(\lg^2(m)/\mathbb{F})$.
- **Memory-checking:** The multiset equality check for each sub-table uses the tuple-equality and multiset-equality arguments from Lemma 7.2.1^o and Lemma 7.2.2^o, each contributing $O(m/\mathbb{F})$.

By a union bound, the total soundness error is:

$$\delta_s \leq O\left(\frac{\lg(k) + c \cdot \lg^2(m) + c \cdot m}{|\mathbb{F}|}\right)$$

Which is negligible for any cryptographically sized field.

Bibliography

1. Setty, S., Thaler, J., & Wahby, R. (2023). Unlocking the lookup singularity with Lasso. (Cryptology ePrint Archive, Paper 2023/1216) Retrieved from <https://eprint.iacr.org/2023/1216>^o
2. Setty, S. (2019). Spartan: Efficient and general-purpose zkSNARKs without trusted setup. (Cryptology ePrint Archive, Paper 2019/550) Retrieved from <https://eprint.iacr.org/2019/550>^o
3. Thaler, J. (2023). Proofs, Arguments, and Zero-Knowledge. (Accessed: 2025-12-18) Retrieved from <https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf>^o
4. Goldwasser, S., Kalai, Y. T., & Rothblum, G. (2008, January). Delegating computation: interactive proofs for muggles. Retrieved from <https://www.microsoft.com/en-us/research/publication/delegating-computation-interactive-proofs-for-muggles/>^o
5. Xie, T., Zhang, J., Zhang, Y., Papamanthou, C., & Song, D. (2019). Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. (Cryptology ePrint Archive, Paper 2019/317) Retrieved from <https://eprint.iacr.org/2019/317>^o
6. Thaler, J. (2013). Time-Optimal Interactive Proofs for Circuit Evaluation. (Cryptology ePrint Archive, Paper 2013/351) Retrieved from <https://eprint.iacr.org/2013/351>^o
7. Blum, M., Evans, W., Gemmell, P., Kannan, S., & Naor, M. (1991). Checking the correctness of memories. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science* (pp. 90–99). <https://doi.org/10.1109/SFCS.1991.185352>^o
8. Pedersen, T. P. (1992). Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In J. Feigenbaum (Ed.), *Advances in Cryptology — CRYPTO '91* (pp. 129–140). Berlin, Heidelberg: Springer Berlin Heidelberg.
9. Maller, M., Bowe, S., Kohlweiss, M., & Meiklejohn, S. (2019). Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings. (Cryptology ePrint Archive, Paper 2019/099) Retrieved from <https://eprint.iacr.org/2019/099>^o
10. Gabizon, A., Williamson, Z. J., & Ciobotaru, O. (2019). PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. (Cryptology ePrint Archive, Paper 2019/953) Retrieved from <https://eprint.iacr.org/2019/953>^o
11. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, P., & Ward, N. (2019). Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. (Cryptology ePrint Archive, Paper 2019/1047) Retrieved from <https://eprint.iacr.org/2019/1047>^o
12. Kate, A., Zaverucha, G. M., & Goldberg, I. (2010). Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security* (Vol. 6477, pp. 177–194). Springer. https://doi.org/10.1007/978-3-642-17373-8_11^o