leanVM

Contents

1 Introduction **4**

2 VM specification **4**

3 Proving primitives **6**

4 Committing the witness **8**
- 4.1 Multilinear stacking . . . 8
- 4.2 Committing to booleans via Ring-Switching . . . 9

5 Arithmetization framework **9**
- 5.1 M3 model . . . 9
- 5.2 Balancing the bus: grand product . . . 10
- 5.3 Proving the grand product . . . 10
  - 5.3.1 Grand-Product argument via GKR . . . 10
  - 5.3.2 Two layers at a time . . . 11
  - 5.3.3 Batching several grand-products . . . 11
- 5.4 Stacking the leaves . . . 11
- 5.5 Table sumcheck . . . 12
- 5.6 Example: proving elliptic curve multiplication . . . 13
  - 5.6.1 Statement . . . 13
  - 5.6.2 Arithmetization . . . 14
  - 5.6.3 The interactive protocol, unrolled . . . 14

6 Bus interactions **15**
- 6.1 VM State . . . 16
- 6.2 The lookup protocol . . . 16
- 6.3 Memory . . . 18
- 6.4 Bytecode . . . 18
- 6.5 The index column . . . 18

7 The instruction tables **18**
- 7.1 `XOR` . . . 19
- 7.2 `MUL_NATIVE` . . . 19
- 7.3 `SET_CONSTANT` . . . 19
- 7.4 `DEREF` . . . 20
- 7.5 `JUMP` . . . 20
- 7.6 `BLAKE2S` . . . 21

8 End-to-end protocol **21**
8.1 Bytecode encoding . . . 21
8.2 Public input . . . 22
8.3 Filling the tables . . . 22
8.4 Fiat Shamir instantiation . . . 22
8.5 The unrolled protocol . . . 22

9 ISA programming **24**
9.1 Hints . . . 24
9.2 Division and inequality . . . 24
9.3 Functions . . . 25
9.4 Loops . . . 25
9.5 Conditionals . . . 25
9.6 Checking that a value is in the subfield K . . . 26
9.7 Range checks . . . 26
9.8 Counter in BLAKE2s . . . 26
9.9 Match statements . . . 26

10 Ethereum use cases **27**
10.1 Recursive aggregation . . . 27
10.1.1 The public input . . . 27
10.1.2 Checking coverage . . . 28
10.1.3 Deferred evaluation claims . . . 28
10.2 Data availability: LeanDA . . . 28
10.2.1 The commitment . . . 28
10.2.2 Codeword membership . . . 29

A Ring switching **30**
A.1 Setting . . . 30
A.2 The reduction . . . 31
A.3 Which maps are sound . . . 31
A.4 The map we draw . . . 32
A.5 Evaluating the weight . . . 33
A.6 Cost . . . 33
A.7 Credits . . . 33

B The polynomial commitment scheme: WHIR /Ligerito **34**
B.1 Preliminaries . . . 34
B.1.1 Codes, interleaving, and list decoding . . . 34
B.1.2 PCS and round-by-round soundness . . . 35
B.2 Reed–Solomon codes over binary fields . . . 35
B.3 The protocol . . . 36
B.4 Round-by-round soundness . . . 37
B.4.1 Proximity tools . . . 37
B.4.2 Proof of the main theorem . . . 38
B.5 Optimizations . . . 40
B.6 The Johnson bound . . . 41

C The Flock protocol **41**
C.1 Statement and commitment . . . 41
C.2 Zerocheck . . . 42
C.3 Lincheck . . . 43
C.3.1 Concluding Lincheck for a native verifier . . . 44
C.3.2 Concluding Lincheck for a recursive verifier . . . 45
C.3.3 Ring switching . . . 45
C.4 Protocol summary . . . 45
C.5 Evaluating the matrices . . . 45
C.5.1 The circuit . . . 46
C.5.2 The R1CS matrices . . . 46
C.5.3 Forward algorithm . . . 46
C.5.4 Transpose algorithm . . . 47
C.6 Soundness analysis . . . 47

D Novel polynomial basis and additive NTT **48**
D.1 Additive domains and subspace polynomials . . . 48
D.2 Novel basis and Reed-Solomon encoding . . . 49
D.3 The additive NTT . . . 49
D.4 Column weights . . . 50
D.5 Duality and codeword membership . . . 51

1 Introduction

Ethereum's consensus layer currently uses BLS signatures, its execution layer ECDSA, and its data availability layer KZG. All three are based on elliptic curves, which do not survive a quantum computer [Sho97]. Hash-based cryptography, both signatures and snarks (the hash-based ones are sometimes called STARKs [BSBHR18]), is a promising candidate for Ethereum's post-quantum transition:

- BLS → XMSS [HBG+18] (using each Ethereum slot as the nonce, statefulness is no longer an issue when attesting: double signing is already a slashable offense). But XMSS has no native aggregation, unlike BLS, so we need a snark to handle it.
- ECDSA → SPHINCS+ [BHK+19]. Drawback: dozens of times bigger. Again, snark-based aggregation is a natural solution to mitigate signature size increase.
- KZG → Reed–Solomon encode each blob, and provide the Merkle root of the codeword. A snark guarantees that the root commits to a correct encoding.

leanVM is a candidate as the snark machinery required above. It's a verifiable virtual machine (commonly a zkVM, for "zero-knowledge virtual machine", though leanVM is not (yet) zk), with a minimalistic ISA (inspired by Cairo [GPR21]). It is based on binary fields, inspired by [DP26] and [BRW26], currently using BLAKE2s [SA15] as hash function.

2 VM specification

Define the following two finite fields:

$$K = \mathbb{F}_{2^{64}} = \mathbb{F}_2[x]/(x^{64} + x^4 + x^3 + x + 1), \qquad E = K[y]/(y^3 + y + 1), \qquad |E| = 2^{192}.$$

And fix a generator g of the multiplicative group $K^\times$, of order $2^{64} - 1$.

An element of K is a polynomial in x of degree below 64, written in hex with the coefficient of x^i in bit i, so $\mathsf{2}$ is x and $\mathsf{3}$ is $x + 1$. An element of E is three of those, its coefficients on $1, y, y^2$.

LeanVM ISA, also known as leanISA, is inspired by [GPR21]. The Virtual Machine takes as input a bytecode (padded to a power of two, $N_{\text{prog}} = 2^{\kappa_{\text{bc}}}$) and some public input $\mathsf{input} \in \{0,1\}^{256} \simeq (\mathsf{input}_0, \ldots, \mathsf{input}_3) \in (\{0,1\}^{64})^4$. The Virtual Machine then initializes its two K-valued registers:

- the program counter $\mathbf{pc} \leftarrow 1_K$
- the frame pointer $\mathbf{fp} \leftarrow 1_K$

Then, initialize the write-once memory: a buffer, of size $N_{\text{mem}} = 2^{\kappa_{\text{mem}}}$, containing E-valued *memory words* (192 bits). The memory is indexed *in the exponent* of g (64 bits). The first memory word is $\mathbf{m}[\mathsf{g}^0] \in E$, the second is $\mathbf{m}[\mathsf{g}^1]$, the third is $\mathbf{m}[\mathsf{g}^2]$, etc. The bytecode addressing follows the same convention. Each memory access beyond $\mathsf{g}^{N_{\text{mem}}-1}$ (resp. $\mathsf{g}^{N_{\text{prog}}-1}$ for the bytecode) is invalid.

The public input fixes the first two memory words $\mathbf{m}[\mathsf{g}^0] = \mathsf{input}_0 + \mathsf{input}_1 \cdot y \in E$ and $\mathbf{m}[\mathsf{g}^1] = \mathsf{input}_2 + \mathsf{input}_3 \cdot y \in E$.

The Virtual Machine execution loop is the following:

1. fetch the instruction inst at address $\mathbf{pc}$.
2. execute inst:

- assert the memory constraints (e.g. $\mathbf{m}[\mathbf{fp} \cdot o_C] = \mathbf{m}[\mathbf{fp} \cdot o_A] + \mathbf{m}[\mathbf{fp} \cdot o_B]$ for XOR)
- update **pc** and **fp** (all instructions except `JUMP` set $\mathbf{pc} \leftarrow \mathbf{pc} \cdot \mathbf{g}$, advancing to the next instruction, and leave $\mathbf{fp} \leftarrow \mathbf{fp}$ unchanged)

3. if **pc** equals $\mathbf{pc}_{\mathsf{final}} := \mathbf{g}^{N_{\text{prog}}-1}$, exit the loop (successful execution). Otherwise go back to 1.

Write-once memory The first time a memory word is set defines its value for the rest of the execution, i.e. it cannot be modified again in the future.

When proving the VM execution via a snark, the memory is committed (by an untrusted prover) at the beginning of the execution (with all the values already filled): every instruction then simply consists in reading a bunch of memory cells, and asserting constraints on them (*write-once memory* is also sometimes called *read-only memory*).

Non determinism For a given pair of bytecode and public input, there is more than one valid execution.

In particular, the memory log-size κ_{mem} is a non deterministic parameter, freely chosen at the beginning of execution, with the constraint $16 \leq \kappa_{\text{mem}} \leq 32$.

Operands and addressing. An instruction is an opcode followed by a list of K-valued operands. An operand is either an *fp-relative reference*, an offset j into the current frame encoded as the $\mathbf{g}$-power $o = \mathbf{g}^j$, or one K-lane of an *immediate* (k_0, k_1, k_2 in `SET_CONSTANT`). With the frame based at $\mathbf{fp} = \mathbf{g}^{\text{base}}$, a reference names the cell

$$\mathbf{m}[\mathbf{fp} \cdot o], \qquad \mathbf{fp} \cdot o \;=\; \mathbf{g}^{\text{base}} \cdot \mathbf{g}^{\,j} \;=\; \mathbf{g}^{\,\text{base}+j}.$$

Instruction set. The machine implements six instructions.

Instruction	Operands	Semantics
`XOR`	$[o_A, o_B, o_C]$	$\mathbf{m}[\mathbf{fp} \cdot o_C] = \mathbf{m}[\mathbf{fp} \cdot o_A] + \mathbf{m}[\mathbf{fp} \cdot o_B]$ (addition in E)
`MUL_NATIVE`	$[o_A, o_B, o_C]$	$\mathbf{m}[\mathbf{fp} \cdot o_C] = \mathbf{m}[\mathbf{fp} \cdot o_A] \cdot \mathbf{m}[\mathbf{fp} \cdot o_B]$ (multiplication in E)
`SET_CONSTANT`	$[o, k_0, k_1, k_2]$	$\mathbf{m}[\mathbf{fp} \cdot o] = k_0 + k_1 y + k_2 y^2$
`DEREF`	$[o_1, o_2, o_3;\ \mathsf{mode}]$	$\mathbf{m}[\mathbf{m}[\mathbf{fp} \cdot o_1] \cdot o_2] = \mathsf{src}(\mathsf{mode})$ (see below)
`JUMP`	$[o_c, o_d, o_f]$	conditional jump (see below)
`BLAKE2S`	$[o_{m_0}, o_{m_1}, o_{m_2}, o_{m_3}, o_{cv}, o_{out}, o_{\text{md}}]$	BLAKE2s compression (see below)

`DEREF` stores, at the dereferenced address $\mathbf{m}[\mathbf{fp} \cdot o_1] \cdot o_2$, a value chosen by a *store mode* $\mathsf{mode} \in \{\texttt{deref\_cell}[o_3], \texttt{deref\_pc}, \texttt{deref\_fp}\}$:

$$\mathsf{src}(\mathsf{mode}) \;=\; \begin{cases} \mathbf{m}[\mathbf{fp} \cdot o_3] & \mathsf{mode} = \texttt{deref\_cell}[o_3] \quad \text{(the local cell at offset } o_3 \in K\text{)}, \\ \mathbf{g}^2 \cdot \mathbf{pc} & \mathsf{mode} = \texttt{deref\_pc} \quad \text{(the current program counter advanced by two)}, \\ \mathbf{fp} & \mathsf{mode} = \texttt{deref\_fp} \quad \text{(the current frame pointer)}. \end{cases}$$

Additionally, it asserts $\mathbf{m}[\mathbf{fp} \cdot o_1] \in K$.

`JUMP` reads a condition $c = \mathbf{m}[\mathbf{fp} \cdot o_c]$, and branches on whether it is zero. When $c \neq 0$ it transfers control, $\mathbf{pc} \leftarrow \mathbf{m}[\mathbf{fp} \cdot o_d]$ and $\mathbf{fp} \leftarrow \mathbf{m}[\mathbf{fp} \cdot o_f]$; when $c = 0$ it defaults to $\mathbf{pc} \leftarrow \mathbf{g} \cdot \mathbf{pc}$ and $\mathbf{fp} \leftarrow \mathbf{fp}$. Additionally, it asserts $\mathbf{m}[\mathbf{fp} \cdot o_c], \mathbf{m}[\mathbf{fp} \cdot o_f], \mathbf{m}[\mathbf{fp} \cdot o_d] \in K$ (unconditionally).

`BLAKE2S` is the standard BLAKE2s compression, consuming a 64-byte message block and a 32-byte chaining value and producing 32 bytes. Each 128-bit chunk occupies one full E memory cell through the canonical embedding $a_0 + a_1 y \mapsto a_0 + a_1 y + 0y^2$; using a cell as a BLAKE2s operand therefore constrains its top limb to zero. The four message chunks are addressed *independently* by references $o_{m_0}, \ldots, o_{m_3}$ (chunk i at address $\mathbf{fp} \cdot o_{m_i}$), while o_{cv} names two *consecutive* chaining-value cells $\mathbf{m}[\mathbf{fp}\cdot o_{cv}], \mathbf{m}[\mathbf{fp}\cdot \mathbf{g}\cdot o_{cv}]$. The 256-bit result is written to the consecutive cells $\mathbf{m}[\mathbf{fp}\cdot o_{out}], \mathbf{m}[\mathbf{fp}\cdot \mathbf{g}\cdot o_{out}]$, and o_{md} names one more cell, so each row accesses nine. That cell $\mathrm{md} = \mathbf{m}[\mathbf{fp} \cdot o_{\mathrm{md}}] = \mathrm{md}_0 + \mathrm{md}_1 y$ packs the standard compression metadata in little-endian order,

$$\mathrm{md} = \mathit{counter}_{64} \parallel \mathit{final}_{32} \parallel \mathit{last\_node}_{32}.$$

3 Proving primitives

Definition 3.1 (Iverson bracket)**.** For a proposition P, $[P] = 1$ if P holds and $[P] = 0$ otherwise.

Definition 3.2 (Multiset)**.** A *multiset* $\{\{\cdots\}\}$ over a set S is a multiplicity function $S \to \mathbb{N}$; $\uplus$ adds multiplicities.

Definition 3.3 (Multilinear extension)**.** A *multilinear* is given by its 2^κ values on the boolean hypercube, a function $f : \{0,1\}^\kappa \to E$. Its *multilinear extension* $\widetilde{f} : E^\kappa \to E$ is the unique polynomial of degree at most one in each variable agreeing with f on $\{0,1\}^\kappa$; we identify the two freely.

Definition 3.4 (Equality indicator)**.** For $X, Y \in E^\kappa$,

$$\mathrm{eq}(X,Y) = \prod_j (X_j Y_j + (1+X_j)(1+Y_j)) = \prod_{j=1}^{\kappa} (1 + X_j + Y_j).$$

eq is multilinear, and $\mathrm{eq}(X,Y) = [X = Y]$ on the boolean hypercube $\{0,1\}^{2\kappa}$.

Definition 3.5 (Inner product over the boolean hypercube)**.** For a, b on $\{0,1\}^\kappa$, $\langle a, b\rangle = \sum_{x\in\{0,1\}^\kappa} a(x)\, b(x)$.

Fact 3.6 (eq carries the interpolation weights)**.** *For an arbitrary point $r \in E^\kappa$,*

$$\widetilde{f}(r) = \sum_{x\in\{0,1\}^\kappa} f(x)\ \mathrm{eq}(r,x) = \langle f,\ \mathrm{eq}(r,\cdot)\rangle.$$

Binding only the first $\ell \le \kappa$ variables gives the same identity with the rest left free: for $r \in E^\ell$ and $x \in \{0,1\}^{\kappa-\ell}$,

$$\widetilde{f}(r,x) = \sum_{u\in\{0,1\}^\ell} \mathrm{eq}(r,u)\, f(u,x),$$

so a partial evaluation is again a multilinear, its values the $\mathrm{eq}(r,\cdot)$-weighted combination of the rows $f(u,\cdot)$.

Lemma 3.7 (Every element outside K is a cubic generator)**.** *For $\alpha \in E \setminus K$, the set $\{1, \alpha, \alpha^2\}$ is a K-basis of E.*

Proof. Degrees multiply in the tower $K \subseteq K(\alpha) \subseteq E$, so $[E : K(\alpha)]\,[K(\alpha) : K] = [E : K] = 3$. That degree is prime and $[K(\alpha) : K] > 1$ because $\alpha \notin K$, hence $[K(\alpha) : K] = 3$ and $K(\alpha) = E$. The minimal polynomial of α over K then has degree 3, so $1, \alpha, \alpha^2$ are K-independent, and three independent elements of a space of K-dimension 3 are a basis. □

Lemma 3.8 (Schwartz–Zippel)**.** *A nonzero $P \in E[X_1, \ldots, X_\kappa]$ of total degree at most d satisfies*

$$\Pr_{r \leftarrow E^\kappa}[P(r) = 0] \;\le\; \frac{d}{|E|}.$$

A nonzero multilinear has total degree at most κ, so it vanishes at a random point with probability at most $\kappa/|E|$.

Proof. Induction on κ. For $\kappa = 1$ a nonzero univariate of degree at most d has at most d roots, so a uniform r_1 is one of them with probability at most $d/|E|$. For $\kappa > 1$ write P in its last variable,

$$P(X_1, \ldots, X_\kappa) = \sum_{k=0}^{d} X_\kappa^k \, Q_k(X_1, \ldots, X_{\kappa-1}),$$

and let k^* be the largest k with $Q_k \neq 0$. Then Q_{k^*} is nonzero in $\kappa - 1$ variables of total degree at most $d - k^*$, so by induction $Q_{k^*}(r_{<\kappa}) = 0$ with probability at most $(d - k^*)/|E|$. Fix $r_{<\kappa}$ with $Q_{k^*}(r_{<\kappa}) \neq 0$: then $X_\kappa \mapsto P(r_{<\kappa}, X_\kappa)$ has degree exactly k^*, hence at most k^* roots, and a uniform r_κ is one with probability at most $k^*/|E|$. Adding the two bounds gives $d/|E|$. □

Corollary 3.9 (Testing an identity at one random point)**.** *Let $A, B \in E[X_1, \ldots, X_\kappa]$ have total degree at most d, and let $r \leftarrow E^\kappa$ be drawn after both are fixed. If $A \neq B$ then*

$$\Pr_r[A(r) = B(r)] \;\le\; \frac{d}{|E|}.$$

Proof. Apply Lemma 3.8 to $A - B$, nonzero of total degree at most d. □

Fact 3.10 (Sumcheck [LFKN92])**.** *Over κ rounds, sumcheck reduces a claim $\sum_{x \in \{0,1\}^\kappa} P(x) = c^{(0)}$, on a polynomial P of degree at most d in each variable, to one evaluation $P(\chi)$ at a random $\chi \in E^\kappa$. Round j carries the running claim $c^{(j-1)}$: the prover sends the univariate*

$$h_j(Y) = \sum_{x \in \{0,1\}^{\kappa - j}} P(\chi_1, \ldots, \chi_{j-1}, Y, x)$$

of degree at most d, the verifier checks $h_j(0) + h_j(1) \stackrel{?}{=} c^{(j-1)}$, samples $\chi_j \leftarrow E$ and takes $c^{(j)} = h_j(\chi_j)$. It ends holding $c^{(\kappa)} = P(\chi)$, for the caller to settle. A false $c^{(0)}$ survives with probability at most $d\kappa/|E|$.

Definition 3.11 (Zerocheck)**.** To prove that a polynomial P vanishes on $\{0,1\}^\kappa$, the verifier samples $r \in E^\kappa$ uniformly and the two parties run sumcheck on

$$\sum_{x \in \{0,1\}^\kappa} P(x)\, \mathrm{eq}(r, x) \stackrel{?}{=} 0.$$

By Fact 3.6 that sum is $\widetilde{g}(r)$, for g the restriction of P to the boolean hypercube: the zero polynomial if P vanishes there, and otherwise a nonzero multilinear, which a random r sends to 0 with probability at most $\kappa/|E|$ (Lemma 3.8).

Lemma 3.12 (Partially fixed zerocheck [DT24])**.** *Let $g : \{0,1\}^\ell \times \{0,1\}^m \to \mathbb{F}_2$ be a nonzero Boolean multilinear. Fix $a \in E^\ell$ such that the values $\{\mathrm{eq}(a, b) : b \in \{0,1\}^\ell\}$ are linearly independent over $\mathbb{F}_2$. If $s \leftarrow E^m$ is sampled uniformly after g is fixed, then*

$$\Pr_s[\widetilde{g}(a, s) = 0] \le \frac{m}{|E|}.$$

Proof. Partially evaluate the first ℓ variables at a. By Fact 3.6, the resulting multilinear $h : \{0,1\}^m \to E$ has values

$$h(y) = \widetilde{g}(a, y) = \sum_{b \in \{0,1\}^\ell} g(b, y)\, \mathrm{eq}(a, b).$$

Because g is nonzero, there is some $y \in \{0,1\}^m$ for which the coefficients $\{g(b, y) : b \in \{0,1\}^\ell\}$ are not all zero. These coefficients lie in $\mathbb{F}_2$, so the assumed $\mathbb{F}_2$-linear independence of the values $\mathrm{eq}(a, b)$ gives $h(y) \neq 0$. Thus h is a nonzero multilinear. Its extension satisfies $\widetilde{h}(s) = \widetilde{g}(a, s)$, and Lemma 3.8 gives the claimed probability bound. □

Two savings on a round message [Gru24]. A round polynomial of degree D travels as its coefficients, and would take $D + 1$ of them, but the round check $h_j(0) + h_j(1) \stackrel{?}{=} s_{j-1}$ already pins one to the incoming claim, so D are enough. The same idea applies to the known factor: a zerocheck's summand carries $\mathrm{eq}(r, \cdot)$, which the verifier can form itself, so only the cofactor travels and it is one degree lower. A zerocheck round on a degree-d cofactor therefore costs d field elements.

Definition 3.13 (Inner-product commitment scheme)**.** A *weighted claim* about a multilinear $g : \{0,1\}^M \to K$ is a pair (W, c) asserting $\langle W, g\rangle = c$, with $c \in E$ and the weight $W : \{0,1\}^M \to E$ *MLE-friendly*: the verifier can evaluate $\widetilde{W}$ anywhere in $\mathrm{poly}(M)$ operations. An *inner-product commitment scheme* commits to g and later proves any such claim,

$$\langle W, g\rangle = \sum_{w \in \{0,1\}^M} W(w)\, g(w) = c.$$

The commitment is over K, the opening over E. An evaluation claim $\widetilde{g}(r) = c$ is the case $W = \mathrm{eq}(r, \cdot)$, by Fact 3.6.

We use WHIR [ACFY25] / Ligerito [NA25] as inner-product commitment scheme (see annex B). The two are essentially the same scheme when WHIR is instantiated with an interleaved code and Ligerito with a Reed–Solomon code.

4 Committing the witness

4.1 Multilinear stacking

Suppose we want to commit to T multilinears $P_1, \ldots, P_T$ over boolean hypercubes of various sizes. We could commit to each independently, at the cost of a large cumulative opening proof. Instead they are concatenated into a large multilinear polynomial q, which we commit to with an inner-product commitment scheme (Definition 3.13), with a protocol to reduce opening claims on P_i to a claim on q. Order them by size, $\kappa_1 \geq \cdots \geq \kappa_T$ variables, and concatenate their values end to end,

$$q \;=\; P_1 \,\|\, P_2 \,\|\, \ldots \,\|\, P_T \,\|\, 0,$$

zero-padded to length 2^M, with $N_{\mathrm{stack}} := \sum_i 2^{\kappa_i}$ the number of placed field elements and $M = \lceil \log_2 N_{\mathrm{stack}} \rceil$. Placed largest first, P_i starts at an offset that is a multiple of 2^{κ_i}; writing $\mathsf{sel}_i \in \{0,1\}^{M-\kappa_i}$ for the high bits of that offset,

$$\widetilde{P}_i(z) \;=\; \widetilde{q}(z, \mathsf{sel}_i) \qquad (z \in \{0,1\}^{\kappa_i}).$$

An evaluation claim $\widetilde{P}_i(\zeta) = c$ is therefore a claim on the stack,

$$\widetilde{q}(\zeta, \mathsf{sel}_i) \;=\; \sum_{w \in \{0,1\}^M} \mathrm{eq}\big((\zeta, \mathsf{sel}_i), w\big)\, \widetilde{q}(w) \;=\; c,$$

a weighted sum over $\widetilde{q}$ with weight $W(w) = \mathrm{eq}((\zeta, \mathsf{sel}_i), w)$.

Batching. By the identity above, the claims the protocol raises on the P_i transfer to a list of claims $\langle W_j, \widetilde{q}\rangle = c_j$ on the stack, with E-valued weights and targets. The verifier then samples $\lambda \in E$, and it remains to prove

$$\Big\langle \sum_j \lambda^{j-1} W_j,\ \widetilde{q} \Big\rangle \stackrel{?}{=} \sum_j \lambda^{j-1} c_j$$

via the PCS (Annex B).

4.2 Committing to booleans via Ring-Switching

Flock's BLAKE2s argument (§7.6) requires a commitment to a Boolean multilinear in $\kappa_{\text{flock}} + \kappa_{\text{skip}}$ variables. Ring switching [DP26] lets the prover pack its low $\kappa_{\text{skip}} = 6$ coordinates into the K-valued multilinear q_{flock} in κ_{flock} variables (Annex A), which occupies one region of the stack in §4.1.

5 Arithmetization framework

The arithmetization is inspired by the M3 model (multi-multiset matching) [Irr]. Rows are unordered, and an instance is accepted exactly when every table's constraints hold and the bus balances.

5.1 M3 model

Tables and columns. A column of height 2^κ is a function $\{0,1\}^\kappa \to K$. A table is a group of columns of equal height. The arithmetization fixes the number of tables N_{tab} and, for each table T_j, its number of columns $N_{\text{col},j}$, its $N_{\text{cons},j}$ constraints and its $N_{\text{flushes},j}$ bus flushes. The table heights are not fixed: the prover announces each table's log height τ_j at the beginning of the proof.

Constraints. The constraints of T_j are $C_{j,1}, \ldots, C_{j,N_{\text{cons},j}}$, each a polynomial over K of degree at most d ($d = 2$ throughout) in the $N_{\text{col},j}$ columns, required to vanish on every row in $\{0,1\}^{\tau_j}$.

The bus. All interactions (between tables) share a single *bus*, a channel carrying tuples of up to $m = 16$ K-elements (shorter ones are zero-padded). Table T_j has $N_{\text{flushes},j}$ flushes $\mathbf{f}_{j,1}, \ldots, \mathbf{f}_{j,N_{\text{flushes},j}}$, each is either a *push* or a *pull*. Each one of the m coordinates in each tuple is a fixed polynomial of degree at most d in the table's columns. A few *boundary* tuples are pushed (resp. pulled) by default to the bus (for instance for the VM state $(\mathsf{ST}, \mathbf{pc}, \mathbf{fp})$, the initial $(\mathsf{ST}, \mathsf{g}^0, \mathsf{g}^0)$ is pushed and the final $(\mathsf{ST}, \mathsf{g}^{N_{\text{prog}}-1}, \mathsf{g}^0)$ is pulled).

Balance. The bus balances when its pushed tuples and its pulled tuples form the same multiset (3.2).

Domain separation. When several interactions share the bus, the first coordinate of every tuple is a *domain separator*, a fixed constant naming its interaction: $\mathsf{ST} = \mathsf{g}^0$ for VM state, $\mathsf{MEM} = \mathsf{g}^1$ for memory, $\mathsf{BC} = \mathsf{g}^2$ for bytecode. (They are needed since all interactions share a common bus.)

5.2 Balancing the bus: grand product

Both multisets (*pulls* and *pushes*) carry the same number of tuples, and padding each with factors of 1, which change no product, makes that number a power of two, 2^μ.

Fingerprint. The verifier samples $\alpha = (\alpha_0, \ldots, \alpha_3) \in E^4$ and maps each tuple $\mathbf{f} \in K^{16}$ to a single E-element:

$$\pi_\alpha(\mathbf{f}) \;=\; \sum_{\iota \in \{0,1\}^4} \mathrm{eq}(\alpha, \iota)\, \mathbf{f}_\iota,$$

the sixteen slots indexed by their bits, low bit first, and smaller tuples padded with 0.

Product check. The verifier samples $\beta \in E$, and checks:

$$\prod_{\mathbf{f} \text{ pushed}} (\beta - \pi_\alpha(\mathbf{f})) \;\stackrel{?}{=}\; \prod_{\mathbf{f} \text{ pulled}} (\beta - \pi_\alpha(\mathbf{f})). \tag{1}$$

Theorem 5.1. *The protocol above has perfect completeness (equal pulled and pushed multisets always make (1) valid) and soundness error $4 \cdot 2^\mu/|E|$ (if the pulled and pushed multisets differ, the probability that (1) passes over uniform $\alpha \in E^4$ and $\beta \in E$).*

Lemma 5.2 (A product identity is a multiset identity)**.** *For a multiset P of tuples write*

$$\Pi_P(A, X) \;=\; \prod_{\mathbf{f} \in P} (X - \pi_A(\mathbf{f})),$$

a polynomial in the five formal variables $A = (A_0, \ldots, A_3)$ and X, with coefficients in K. The two sides of (1) are $\Pi_P(\alpha, \beta)$ for P the pushed multiset and the pulled one.

For finite multisets P, Q of width-m tuples over K, $\Pi_P = \Pi_Q$ if and only if $P = Q$.

Proof of Lemma 5.2. TODO □

Proof of Theorem 5.1. Completeness. Equal pulled and pushed multisets make (1) hold for every α and β, so completeness is immediate.

Soundness. Different pulled and pushed multisets make $\Pi_P \neq \Pi_Q$, by Lemma 5.2, and (1) tests them at the one point $(\alpha_0, \ldots, \alpha_3, \beta)$. Both sides have degree at most $4 \cdot 2^\mu$ in $(\alpha_0, \ldots, \alpha_3, \beta)$, since there are at most 2^μ factors, of the form $\beta - \pi_\alpha(\mathbf{f})$, each of total degree at most 4. Applying Corollary 3.9 gives a soundness error of $4 \cdot 2^\mu/|E|$. □

5.3 Proving the grand product

Take one side of (1). Its tuples give its 2^μ leaves $V_0(x) = \beta - \pi_\alpha(\mathbf{f}_x)$, the ones past the last tuple being 1. Multiply the leaves two by two, then multiply the results two by two, and so on: after μ layers one value is left, the root $P = \prod_x V_0(x)$. Layer 0 is the leaves, layer i has $2^{\mu-i}$ nodes, and layer μ is the root. The prover announces P. GKR proves it by walking back down the tree, turning a claim about one layer into a claim about the layer below, until it reaches the leaves and §5.4 takes over.

5.3.1 Grand-Product argument via GKR

Write $\widetilde{V}_i : E^{\mu-i} \to E$ for the multilinear extension of layer i. Every node of layer i is the product of its two children,

$$V_i(x) = V_{i-1}(0, x)\, V_{i-1}(1, x), \qquad x \in \{0,1\}^{\mu-i},$$

so by Fact 3.6 a claim on $\widetilde{V}_i$ at a point r is a sum over the hypercube,

$$\widetilde{V}_i(r) = \sum_{x \in \{0,1\}^{\mu-i}} \mathrm{eq}(r, x)\, \widetilde{V}_{i-1}(0, x)\, \widetilde{V}_{i-1}(1, x),$$

which sumcheck (Fact 3.10) reduces to the two values $\widetilde{V}_{i-1}(0, \chi)$ and $\widetilde{V}_{i-1}(1, \chi)$ at one random χ. The prover sends them and a fresh challenge u combines them multilinearly, leaving the single claim $\widetilde{V}_{i-1}(u, \chi)$: the same kind of claim, one layer lower. The walk starts at the root, where the claim is the announced P itself, and μ such steps end on the leaves.

5.3.2 Two layers at a time

For efficiency, we contract two binary levels at once, a node being the product of its four grandchildren:

$$V_i(x) = \prod_{a,b \in \{0,1\}} V_{i-2}(a, b, x).$$

The same sumcheck then reduces $\widetilde{V}_i(r)$ to the four values $\widetilde{V}_{i-2}(a, b, \chi)$, which the prover sends and two fresh challenges (u_0, u_1) combine into one claim $\widetilde{V}_{i-2}(u_0, u_1, \chi)$, two layers down.

If μ is odd, GKR first processes the root-most binary layer, which has no sumcheck rounds, and then proceeds entirely in radix four.

5.3.3 Batching several grand-products

For recursion friendliness, the N_{side} grand products, numbered $s = 1, \ldots, N_{\mathrm{side}}$, are proven in one pass, at one sumcheck per layer instead of N_{side}.

Pad every tree with 1-leaves up to the depth μ of the deepest. The verifier then samples at every layer a combiner $\lambda \in E$, and each layer runs a single sumcheck on the random linear combination of the N_{side} identities,

$$\sum_{s=1}^{N_{\mathrm{side}}} \lambda^{s-1} \widetilde{V}_i^{s}(r) = \sum_{x \in \{0,1\}^{\mu-i}} \mathrm{eq}(r, x) \sum_{s=1}^{N_{\mathrm{side}}} \lambda^{s-1} \prod_{a,b \in \{0,1\}} \widetilde{V}_{i-2}^{s}(a, b, x)$$

All N_{side} trees leave the pass on claims at one shared terminal point ζ.

5.4 Stacking the leaves

The GKR grand-product argument reduces each side of (1) to a single evaluation $\widetilde{V}_0(\zeta)$ of its *leaves*, the product's factors $\beta - \pi_\alpha(\mathbf{f})$, one per bus tuple $\mathbf{f}$.

Flush blocks. The leaves split into *blocks*, one per flush rule and one per boundary set. A block of a 2^κ-row table holds 2^κ leaves; the row-z leaf is $\beta - \sum_\iota \mathrm{eq}(\alpha, \iota)\, c_\iota(z)$, with $c_\iota(z)$ slot ι of the row's tuple. Each c_ι is a polynomial of degree at most $d = 2$ in the columns (at the corresponding row) (§5.1).

Layout. Order the blocks largest first at aligned offsets, padding to 2^μ leaves with 1s, which leave the product unchanged. This is the stacking of §4.1 applied to blocks: block b, of 2^{κ_b} leaves, occupies the subcube $\{(z, \mathsf{sel}_b) : z \in \{0,1\}^{\kappa_b}\}$ for a fixed selector sel_b (that depends only on the table, bytecode and memory sizes).

Decomposition. Split $\zeta = (\zeta^b_{\text{lo}}, \zeta^b_{\text{hi}})$ at block b's boundary. Each block is a subcube, so the leaf MLE splits into a public selector times the block's own MLE, plus what the padding contributes:

$$\widetilde{V}_0(\zeta) \;=\; \sum_b \mathrm{eq}(\mathsf{sel}_b, \zeta^b_{\text{hi}})\Big(\beta - \sum_\iota \mathrm{eq}(\alpha, \iota)\, \widetilde{c}_{b,\iota}(\zeta^b_{\text{lo}})\Big) \;+\; \Big(1 + \sum_b \mathrm{eq}(\mathsf{sel}_b, \zeta^b_{\text{hi}})\Big). \tag{2}$$

That last term is the padding's weight, for the final $\ldots 1111$.

Settling it. Fix one side s. The boundary tuples (§5.1) and each array's seed and finalization (§6.2) belong to no table. Their entries have degree one: a constant, the index column (§6.5), a public component, or one committed column. The prover sends, for each such committed multilinear, its evaluation at ζ^b_{lo}. These claims are then deferred to the PCS (§4.1). The verifier finally evaluates each contribution, and subtracts them, with the corresponding padding term, from the claimed value of $\widetilde{V}^s_0(\zeta)$. Call the remainder rem_s: it remains to prove an equation similar to (2), summing only on the flush blocks from tables.

Let $\mathcal{B}$ be the blocks of T_j's flushes in this leaf vector, one per flush rule (§5.1). Each holds one leaf per row of T_j, so each is 2^{τ_j} tall and all of them split ζ at the same place, $\zeta^b_{\text{lo}} = \zeta_{<\tau_j}$ and $\zeta^b_{\text{hi}} = \zeta_{\geq\tau_j}$, differing only through their selector. Exchanging the two sums,

$$\sum_{b\in\mathcal{B}} \mathrm{eq}(\mathsf{sel}_b, \zeta^b_{\text{hi}}) \sum_{x\in\{0,1\}^{\tau_j}} \mathrm{eq}(\zeta_{<\tau_j}, x)\Big(\beta - \sum_\iota \mathrm{eq}(\alpha,\iota)\, c_{b,\iota}(x)\Big) \;=\; \sum_{x\in\{0,1\}^{\tau_j}} \mathrm{eq}(\zeta_{<\tau_j}, x)\, B^s_j(x),$$

$$B^s_j \;=\; \sum_{b\in\mathcal{B}} \mathrm{eq}(\mathsf{sel}_b, \zeta^b_{\text{hi}})\Big(\beta - \sum_\iota \mathrm{eq}(\alpha,\iota)\, c_{b,\iota}\Big).$$

B^s_j has degree at most d in T_j's columns, and the verifier builds its coefficients from α, β and the selectors. What remains to prove is

$$\sum_j \sum_{x\in\{0,1\}^{\tau_j}} \mathrm{eq}(\zeta_{<\tau_j}, x)\, B^s_j(x) \;\overset{?}{=}\; \mathsf{rem}_s.$$

There is one such equation per side. Nothing is sent for these blocks. Each constraint of T_j owes a sum of the same shape, so one sumcheck over the tables' rows proves them all (§5.5).

5.5 Table sumcheck

Proving constraints via zerocheck. $C_{j,i}$ is a polynomial in T_j's columns, so it is a function $\{0,1\}^{\tau_j} \to K$ that must vanish on every row. Following Fact 3.6, this can be proved by sampling $r \in E^{\tau_j}$ uniformly at random and running sumcheck on:

$$\widetilde{C_{j,i}}(r) \;=\; \sum_{x\in\{0,1\}^{\tau_j}} \mathrm{eq}(r, x)\, C_{j,i}(x) \;\overset{?}{=}\; 0$$

Zerocheck point recycling. Nothing forces r to be sampled just before the zerocheck: it may come from a previous sub-protocol, as long as it results from uniform sampling occurring after the table columns are committed. We take $r = \zeta_{<\tau_j}$, the first τ_j coordinates of the GKR terminal point ζ (§5.3). Validity of the constraints is thus reduced to:

$$\sum_{x\in\{0,1\}^{\tau_j}} \mathrm{eq}(\zeta_{<\tau_j}, x)\, C_{j,i}(x) \;\overset{?}{=}\; 0, \qquad \text{for } j \in \{1,\ldots,N_{\text{tab}}\},\ i \in \{1,\ldots,N_{\text{cons},j}\}. \tag{3}$$

The bus claims join in. §5.4 left one per side, at that same point,

$$\sum_{j=1}^{N_{\text{tab}}} \sum_{x\in\{0,1\}^{\tau_j}} \mathrm{eq}(\zeta_{<\tau_j}, x)\, B_j^s(x) \stackrel{?}{=} \mathsf{rem}_s, \qquad \text{for } s \in \{1,\dots,N_{\text{side}}\}. \tag{4}$$

Both (3) and (4) are built from $\mathrm{eq}(\zeta_{<\tau_j},\cdot)$-weighted sums over one table's rows of a polynomial of degree at most d in its columns: we batch them in a single sumcheck.

Back-loaded batched sumcheck. The verifier samples $\xi \in E$ and gives each claim its own power of it: ξ^{o_j+i-1} for $C_{j,i}$, with $o_j = N_{\text{cons},1} + \cdots + N_{\text{cons},j-1}$ numbering the constraints table by table, and then $\xi^{N_{\text{cons}}+s-1}$ for side s, with $N_{\text{cons}} = \sum_j N_{\text{cons},j}$ the constraint count. A sumcheck runs over one cube, so write $\tau_{\max} = \max_j \tau_j$, let $X_{<\tau_j} = (X_0,\dots,X_{\tau_j-1})$ be the variables T_j uses, and multiply in the ones it does not:

$$F(X_0,\dots,X_{\tau_{\max}-1}) = \sum_{j=1}^{N_{\text{tab}}} \mathrm{eq}(\zeta_{<\tau_j}, X_{<\tau_j})\Big(\sum_{i=1}^{N_{\text{cons},j}} \xi^{o_j+i-1} C_{j,i} + \sum_{s=1}^{N_{\text{side}}} \xi^{N_{\text{cons}}+s-1} B_j^s\Big)(X_{<\tau_j}) \prod_{k=\tau_j}^{\tau_{\max}-1} X_k.$$

Since $\sum_{x\in\{0,1\}^m} x_0\cdots x_{m-1} = 1$, the padded variables contribute a factor 1, so summing F over $\{0,1\}^{\tau_{\max}}$ gives back what each table owes. We thus run a single sumcheck:

$$\sum_{x\in\{0,1\}^{\tau_{\max}}} F(x) \stackrel{?}{=} \sum_{s=1}^{N_{\text{side}}} \xi^{N_{\text{cons}}+s-1} \mathsf{rem}_s,$$

the verifier computing the right side itself. Each summand of F has individual degree $d+1=3$, the constraint contributing d and the eq factor one, so the soundness error is $(3\tau_{\max} + N_{\text{side}} + N_{\text{cons}})/|E|$: $3\tau_{\max}$ for the rounds (Fact 3.10) and $N_{\text{side}} + N_{\text{cons}}$ for the ξ-batching (Corollary 3.9). Rounds bind $X_{\tau_{\max}-1}$ first, so T_j joins only at round $\tau_{\max} - \tau_j$: hence *back-loaded*. The end yields one claim per column, T_j's at $\chi_{<\tau_j}$, settled by the PCS opening (§4.1).

5.6 Example: proving elliptic curve multiplication

5.6.1 Statement

The claim. Over K, with $\mathsf{g} = 2$, fix the elliptic curve $t^2 + e\,t = e^3 + \mathsf{g}$ and the two public points

$$G = (e_G, t_G) = (1,\ \texttt{9d7b84b595b2e2e6}),$$
$$Q = (e_Q, t_Q) = (\texttt{bbde619e481ba9dd},\ \texttt{c7dc5bdf1e3477f9}).$$

The group law gives:

$$[2](e,t) = (e',\ e^2 + (w+1)\,e'), \qquad w = e + t/e, \qquad e' = w^2 + w,$$
$$(e,t) + G = (e',\ w\,(e+e') + e' + t), \qquad w = (t+t_G)/(e+e_G), \qquad e' = w^2 + w + e + e_G.$$

The statement is: $Q = [66067]G$. We prove it via a double-and-add walk. Since $66067 = 2^{16} + 2^9 + 2^4 + 2 + 1$, there are sixteen doublings and four additions of G.

5.6.2 Arithmetization

The tables.

- $T_1 = \texttt{DOUBLE}$, height 2^4.

  Columns: $\mathsf{tag}, e, t, w, e'$.

  Constraints: $C_{1,1} : w\,e = e^2 + t$ and $C_{1,2} : e' = w^2 + w$.

  Flushes: PULL (tag, e, t) and PUSH $(\mathsf{tag}^2,\ e',\ e^2 + (w+1)\,e')$.

- $T_2 = \texttt{ADD}$, height 2^2.

  Columns: $\mathsf{tag}, e, t, w, e', v$.

  Constraints: $C_{2,1} : v\,(e{+}e_G) = 1$, $C_{2,2} : w\,(e{+}e_G) = t{+}t_G$ and $C_{2,3} : e' = w^2{+}w{+}e{+}e_G$.

  Flushes: PULL (tag, e, t) and PUSH $(\mathsf{g}\cdot\mathsf{tag},\ e',\ w\,(e+e') + e' + t)$.

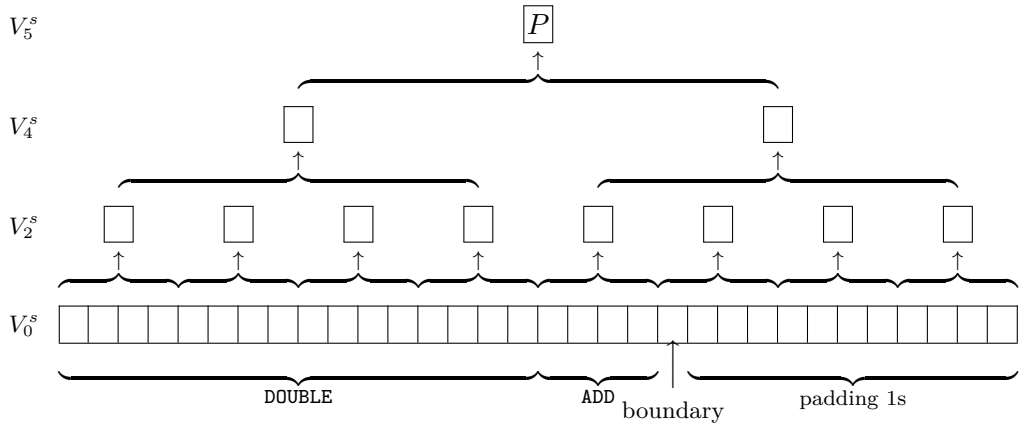
The column v in T_2 is required to prevent a row at the point G, where $e{+}e_G = 0$ and $w\,(e{+}e_G) = t{+}t_G$ holds for any w. So $N_{\text{tab}} = 2$ and $N_{\text{cons}} = 5$.

Boundary flushes. PUSH (g, e_G, t_G) and PULL $(\mathsf{g}^{66067}, e_Q, t_Q)$.

The existence of T_1 and T_2 verifying the constraints, and balancing the bus (including the boundary flushes) proves the statement.

5.6.3 The interactive protocol, unrolled

1. **Prover.** Commits the eleven columns. In this example, the two log heights are fixed, $\tau_1 = 4$ and $\tau_2 = 2$.
2. **Verifier.** Samples and sends $\alpha_0, \alpha_1, \alpha_2, \alpha_3$ and β.
3. **Prover.** Builds the leaves $\beta - \pi_\alpha(\mathbf{f})$, $16 + 4 + 1 = 21$ per side (PUSH / PULL), and pads to $\mu = 5$ with eleven 1s. The two sides share the same layout (largest blocks first):



   The boundary block is leaf 20, and the three selectors are (0), $(0, 0, 1)$ and $(0, 0, 1, 0, 1)$.
4. **Prover.** Sends one root P shared by both sides.
5. **Prover & Verifier.** For each side, run Grand-Product GKR on the tree above: μ is odd, so a binary layer first, then two radix-four steps. The sumchecks of each side are batched together, with final point $\zeta \in E^5$.

6. **Verifier.** Evaluates leaf 20, whose entries are public, and the weight of the eleven padding leaves. Subtracting both from $\widetilde{V}_0^s(\zeta)$ leaves $\mathsf{rem}_{\text{PUSH}}$ and $\mathsf{rem}_{\text{PULL}}$.
7. **Verifier.** Samples and sends the batching scalar ξ.
8. **Prover & Verifier.** Seven claims are now open, and every one of them is a sum over one table's rows against its own weight $\mathrm{eq}(\zeta_{<\tau_j}, \cdot)$. The five constraints have to vanish, and the two sides have to return what step 6 left:

$$\sum_{x\in\{0,1\}^{\tau_j}} \mathrm{eq}(\zeta_{<\tau_j}, x)\, C_{j,i}(x) \overset{?}{=} 0 \qquad \text{for the five } C_{j,i},$$

$$\sum_{j=1}^{2} \sum_{x\in\{0,1\}^{\tau_j}} \mathrm{eq}(\zeta_{<\tau_j}, x)\, B_j^s(x) \overset{?}{=} \mathsf{rem}_s \qquad \text{for } s \in \{\text{PUSH}, \text{PULL}\},$$

where B_j^s is what T_j owes on side s. Writing a column's value at row x as $e(x)$ and so on:

$$B_1^{\text{PUSH}}(x) = \mathrm{eq}((0), \zeta_{\geq 4})\Big(\beta - \pi_\alpha(\mathsf{tag}(x)^2,\ e'(x),\ e(x)^2 + (w(x)+1)\, e'(x))\Big),$$
$$B_1^{\text{PULL}}(x) = \mathrm{eq}((0), \zeta_{\geq 4})\Big(\beta - \pi_\alpha(\mathsf{tag}(x),\ e(x),\ t(x))\Big),$$
$$B_2^{\text{PUSH}}(x) = \mathrm{eq}((0,0,1), \zeta_{\geq 2})\Big(\beta - \pi_\alpha(\mathsf{g}\cdot\mathsf{tag}(x),\ e'(x),\ w(x)\,(e(x)+e'(x)) + e'(x) + t(x))\Big),$$
$$B_2^{\text{PULL}}(x) = \mathrm{eq}((0,0,1), \zeta_{\geq 2})\Big(\beta - \pi_\alpha(\mathsf{tag}(x),\ e(x),\ t(x))\Big).$$

They run over different cubes, $\{0,1\}^4$ for `DOUBLE` and $\{0,1\}^2$ for `ADD`, so `ADD`'s part is multiplied by X_2X_3, the two variables it does not use, whose sum over $\{0,1\}^2$ is 1 and so leaves its own sum untouched:

$$F(X_0,\dots,X_3) = \mathrm{eq}(\zeta_{<4}, (X_0,\dots,X_3))\Big(\xi^0 C_{1,1} + \xi^1 C_{1,2} + \xi^5 B_1^{\text{PUSH}} + \xi^6 B_1^{\text{PULL}}\Big)$$
$$+ \mathrm{eq}(\zeta_{<2}, (X_0,X_1))\Big(\xi^2 C_{2,1} + \xi^3 C_{2,2} + \xi^4 C_{2,3} + \xi^5 B_2^{\text{PUSH}} + \xi^6 B_2^{\text{PULL}}\Big) X_2X_3.$$

One sumcheck over $\{0,1\}^4$ then proves all seven claims at once:

$$\sum_{x\in\{0,1\}^4} F(x) \overset{?}{=} \xi^5 \mathsf{rem}_{\text{PUSH}} + \xi^6 \mathsf{rem}_{\text{PULL}}.$$

9. **Prover.** Sends its eleven column values: $\widetilde{\mathsf{tag}}, \widetilde{e}, \widetilde{t}, \widetilde{w}, \widetilde{e'}$ at $\chi = (\chi_0,\dots,\chi_3)$ for `DOUBLE`, and $\widetilde{\mathsf{tag}}, \widetilde{e}, \widetilde{t}, \widetilde{w}, \widetilde{e'}, \widetilde{v}$ at $\chi_{<2} = (\chi_0, \chi_1)$ for `ADD`.
10. **Verifier.** Rebuilds every $C_{j,i}$ and B_j^s from those eleven values, forms $F(\chi)$, and checks it against the sumcheck's last round.
11. **Prover & Verifier.** Verify each of the eleven column claims against their initial commitment.

6 Bus interactions

The machine runs three interactions on the bus, differentiated by their domain separators (§5.1): the VM state, the read-only memory, and the public bytecode. The last two are lookups using offline memory checking [BEG+94].

6.1 VM State

The state interaction uses domain separator $\mathsf{ST} = \mathsf{g}^0$, and carries the 2 registers, program counter and frame pointer: $(\mathsf{ST}, \mathbf{pc}, \mathbf{fp})$. Each instruction row pulls its current state and pushes its successor $(\mathsf{ST}, \mathrm{next}(\mathbf{pc}), \mathrm{next}(\mathbf{fp}))$; the bus is preseeded by pushing the initial state $(\mathsf{ST}, \mathbf{pc}_{\mathrm{initial}}, \mathbf{fp}_{\mathrm{initial}}) = (\mathsf{ST}, \mathsf{g}^0, \mathsf{g}^0)$ and pulling the final state $(\mathsf{ST}, \mathbf{pc}_{\mathrm{final}}, \mathbf{fp}_{\mathrm{final}}) = (\mathsf{ST}, \mathsf{g}^{N_{\mathrm{prog}}-1}, \mathsf{g}^0)$ (boundary conditions).

Proposition 6.1 (Balance implies the existence of a valid execution)**.** *Let $\mathcal{R}$ be a finite multiset of rows, each pulling a state $\mathsf{pull}(r)$ and pushing a state $\mathsf{push}(r)$, and suppose the state channel balances:*

$$\{\{\mathsf{push}(r) : r \in \mathcal{R}\}\} \uplus \{\{(\mathbf{pc}_{\mathrm{initial}}, \mathbf{fp}_{\mathrm{initial}})\}\} \;=\; \{\{\mathsf{pull}(r) : r \in \mathcal{R}\}\} \uplus \{\{(\mathbf{pc}_{\mathrm{final}}, \mathbf{fp}_{\mathrm{final}})\}\}.$$

Then $\mathcal{R}$ splits into rows $r_1, \ldots, r_L$ with

$$\mathsf{pull}(r_1) = (\mathbf{pc}_{\mathrm{initial}}, \mathbf{fp}_{\mathrm{initial}}), \qquad \mathsf{push}(r_i) = \mathsf{pull}(r_{i+1}), \qquad \mathsf{push}(r_L) = (\mathbf{pc}_{\mathrm{final}}, \mathbf{fp}_{\mathrm{final}}),$$

and rows forming closed walks.

Proof. Build the directed multigraph G on the states: one edge $\mathsf{pull}(r) \to \mathsf{push}(r)$ per row, plus one edge e from $(\mathbf{pc}_{\mathrm{final}}, \mathbf{fp}_{\mathrm{final}})$ to $(\mathbf{pc}_{\mathrm{initial}}, \mathbf{fp}_{\mathrm{initial}})$.

Fix a state x. Its in-degree in G counts the rows with $\mathsf{push}(r) = x$, plus e if $x = (\mathbf{pc}_{\mathrm{initial}}, \mathbf{fp}_{\mathrm{initial}})$. Its out-degree counts the rows with $\mathsf{pull}(r) = x$, plus e if $x = (\mathbf{pc}_{\mathrm{final}}, \mathbf{fp}_{\mathrm{final}})$. Those two counts are the multiplicities of x on the two sides of the hypothesis, so they are equal: G is balanced at every state.

A finite balanced multigraph is a union of edge-disjoint closed walks: follow unused edges from any vertex, which balance always permits on entering one, until a vertex repeats; remove the closed walk found and repeat. Deleting e from the closed walk that contains it leaves $r_1, \ldots, r_L$. □

Remark 6.2. Tolerating closed walks is what lets a table be filled to a power of two (§8.3).

6.2 The lookup protocol

We use a similar idea to [DP23, Protocol. 4.27].

Setting. A *lookup array* $\mathbf{L}$ holds 2^κ entries, entry $\mathbf{L}[i]$ being a tuple of N_{comp} elements of K. Its *components* $\mathbf{L}_0, \ldots, \mathbf{L}_{N_{\mathrm{comp}}-1}$ are the multilinears of height 2^κ holding one coordinate of the entry each: they are either committed or public. A *read* is a row of a table (§5.1) naming an address $a \in K$ and a value $v \in K^{N_{\mathrm{comp}}}$, and is correct when $a = \mathsf{g}^i$ and $v = \mathbf{L}[i]$ for some $i < 2^\kappa$.

Tuple and columns. The bus interaction uses a tuple of the form $(\mathsf{sep}, \mathit{addr}, \mathit{count}, \mathit{val})$: a domain separator naming the array, the address, the count, then the entry in coordinates $3, \ldots, N_{\mathrm{comp}} + 2$. Every higher coordinate is zero, so $N_{\mathrm{comp}} \le m - 3$ for the bus width m (§5.1). Take every part of a read to be a committed K-valued column of the reading table: its address, its count, and the N_{comp} components of the value read. Committed on top of $\mathbf{L}$ is one finalize-count K-valued column $\mathit{count}^{\mathrm{fin}}$ of height 2^κ.

Flush rules. With $A[i]$ the number of reads of address g^i:

- **Seed:** for each $i < 2^\kappa$, PUSH $(\mathsf{sep}, \mathsf{g}^i, 1, \mathbf{L}[i])$. A boundary block owned by no table, its address coordinate the index column of §6.5.
- **Read** of v at a with count *count*: PULL $(\mathsf{sep}, a, \mathit{count}, v)$ and PUSH $(\mathsf{sep}, a, \mathsf{g} \cdot \mathit{count}, v)$, both flushes of the reading row.
- **Finalize:** for each i, PULL $(\mathsf{sep}, \mathsf{g}^i, \mathit{count}^{\mathrm{fin}}[i], \mathbf{L}[i])$. An honest prover commits $\mathit{count}^{\mathrm{fin}}[i] = \mathsf{g}^{A[i]}$, the count the address has reached.

Completeness. An honest prover gives the t-th read of g^i the count g^t and commits $\mathit{count}^{\mathrm{fin}}[i] = \mathsf{g}^{A[i]}$: both sides then carry the counts $\mathsf{g}^0, \ldots, \mathsf{g}^{A[i]}$ at that address, so the bus balances.

Lemma 6.3 (Invariant count multisets)**.** *Let $\mathcal{N}$ be a finite multiset over K with $\mathsf{g} \cdot \mathcal{N} = \mathcal{N}$. If $|\mathcal{N}| < 2^{64} - 1$, every element of $\mathcal{N}$ is 0.*

Proof. Let $\mathrm{mult} : K \to \mathbb{N}$ be the multiplicity function of $\mathcal{N}$ (Definition 3.2). Multiplication by g is a bijection of K and $\mathsf{g} \cdot \mathcal{N}$ has multiplicity function $c \mapsto \mathrm{mult}(\mathsf{g}^{-1}c)$, so the hypothesis reads $\mathrm{mult}(\mathsf{g}c) = \mathrm{mult}(c)$ for all c. Every nonzero c is a power of g, so iterating this gives one value mult_1 on all of $K^\times$, which has $2^{64} - 1$ elements. Then $|\mathcal{N}| = \mathrm{mult}(0) + \mathrm{mult}_1(2^{64} - 1) < 2^{64} - 1$ forces $\mathrm{mult}_1 = 0$. □

Theorem 6.4 (Lookup correctness)**.** *Let the flush rules above be the only flushes carrying $\mathbf{L}$'s separator, and write (a_j, c_j, v_j) for the address, count and value that read j evaluates to. If the bus balances, every $c_j \neq 0$, and there are fewer than $2^{64} - 1$ reads, then every read is correct: $(a_j, v_j) = (\mathsf{g}^i, \mathbf{L}[i])$ for some $i < 2^\kappa$.*

Proof. Fix a read pair (a, v) that is *not* an entry of the array, so $(a, v) \neq (\mathsf{g}^i, \mathbf{L}[i])$ for every $i < 2^\kappa$. Balance equates the multiset of pushed tuples with the multiset of pulled ones; keep in each only the tuples carrying $\mathbf{L}$'s separator, address a and entry v, and the two are still equal. A seed and a finalization sit at the pairs $(\mathsf{g}^i, \mathbf{L}[i])$, different from (a, v), so neither survives. A read at (a, v) does, on both sides at once: it pulls its count c and pushes $\mathsf{g}c$. So the survivors are exactly the reads at (a, v).

Writing $\mathcal{N}$ for the multiset of counts of the reads at (a, v), balance therefore says $\mathsf{g} \cdot \mathcal{N} = \mathcal{N}$. There are fewer than $2^{64} - 1$ of them, so Lemma 6.3 makes every count in $\mathcal{N}$ zero; every count is nonzero by hypothesis, so $\mathcal{N}$ is empty. No read happens at such an invalid (a, v).

□

Corollary 6.5 (Addresses are in range)**.** *Under the same hypotheses every read address is an entry address g^i, $i < 2^\kappa$: no read at any other address, 0 included, occurs in an accepted proof (except with negligible soundness error).*

Nothing checks the finalize counts. $\mathit{count}^{\mathrm{fin}}$ appears nowhere in Theorem 6.4 or its proof. A wrong value (such as 0 for instance) cannot make a bad read pass; it can only unbalance the bus at that entry, making the proof invalid.

Meeting the hypotheses. Grand-product GKR checks balance (§5.2). It remains to check that all counts are nonzero, and the $N_{\mathrm{read}} < 2^{64} - 1$ read bound, as follows.

The count product. All counts are nonzero exactly when their product is. The prover therefore sends that product, one E-element, and the verifier checks it is nonzero. It is proven like any other grand product, by GKR over the stacking of every count column (§5.4), batched with the two other GKRs of the bus (push / pull) (§5.3).

Instance caps. The verifier checks the announced sizes against public caps (§8.5): at most 2^{32} rows per table, at most 2^{32} memory cells (§6.3) and at most 2^{32} bytecode instructions (§6.4). A row reads at most ten times (BLAKE2S: nine cells and its bytecode entry), so the six tables flush $N_{\text{read}} \leq 10 \cdot 6 \cdot 2^{32} < 2^{38}$ reads, far under the $2^{64} - 1$ that Lemma 6.3 requires.

Relaxing the hypotheses. A bus entry may be any polynomial of degree at most $d = 2$ in the row's columns (§5.1), so none of the three (address, count, value) need be committed. Example 1: we don't need to commit memory addresses of the form $\mathbf{fp} \cdot o$ (§6.3) when $\mathbf{fp}$ and o are already committed. Example 2: once the pull counter *count* is committed, no need to commit to the corresponding push counter $\mathsf{g} \cdot \mathit{count}$.

6.3 Memory

The memory is an array of $N_{\text{mem}} = 2^{\kappa_{\text{mem}}}$ cells, $16 \leq \kappa_{\text{mem}} \leq 32$ (§2). Bus interaction uses domain separation $\mathsf{sep} = \mathsf{MEM} = \mathsf{g}^1$. A memory word is 192-bit (i.e. one E-element), so $N_{\text{comp}} = 3$, memory is committed as 3 K-valued multilinears in κ_{mem} variables $\mathbf{m}_0, \mathbf{m}_1, \mathbf{m}_2$.

6.4 Bytecode

The bytecode is an array of $N_{\text{prog}} = 2^{\kappa_{\text{bc}}}$ instructions, $\kappa_{\text{bc}} \leq 32$ (§8.1). Bus interaction uses domain separation $\mathsf{sep} = \mathsf{BC} = \mathsf{g}^2$. An instruction is an opcode plus seven operands, so $N_{\text{comp}} = 8$: with the separator, the address and the count ahead of them, this is the widest entry on the bus at 11 coordinates, which is what forces the four index bits, hence the $m = 16$ slots of §5.1. Coordinate 3 is the opcode, naming the instruction:

$$\mathsf{op}_{\mathsf{XOR}} = \mathsf{g}^0, \quad \mathsf{op}_{\mathsf{MUL}} = \mathsf{g}^1, \quad \mathsf{op}_{\mathsf{SET}} = \mathsf{g}^2, \quad \mathsf{op}_{\mathsf{DRF}} = \mathsf{g}^3, \quad \mathsf{op}_{\mathsf{JMP}} = \mathsf{g}^4, \quad \mathsf{op}_{\mathsf{B2S}} = \mathsf{g}^5.$$

The program is public, i.e. not committed, the verifier can evaluate it directly.

6.5 The index column

The seed and finalization blocks run over every address $\mathsf{g}^0, \ldots, \mathsf{g}^{2^\kappa - 1}$, so the verifier needs the extension of that *index column* idx at the point ζ (§5.4). Writing w for the bits of i, $\mathsf{g}^i = \prod_k (\mathsf{g}^{2^k})^{w_k}$, whose k-th factor is $1 + w_k(1 + \mathsf{g}^{2^k})$, which gives the efficient MLE formula:

$$\widetilde{\mathrm{idx}}(\zeta) \;=\; \prod_{k=0}^{\kappa-1} \left(1 + \zeta_k \left(1 + \mathsf{g}^{2^k}\right)\right),$$

7 The instruction tables

Each instruction has its own table.

Write $\mathsf{read}\,(\mathsf{sep}, \mathit{addr}, \mathit{count}, \mathit{val}_0, \mathit{val}_1, \ldots)$ for the pair PULL $(\mathsf{sep}, \mathit{addr}, \mathit{count}, \mathit{val}_0, \mathit{val}_1, \ldots)$, PUSH $(\mathsf{sep}, \mathit{addr}, \mathsf{g} \cdot \mathit{count}, \mathit{val}_0, \mathit{val}_1, \ldots)$: every flush below but the state's.

7.1 XOR

XOR $[o_A, o_B, o_C]$ asserts $\mathbf{m}[\mathbf{fp} \cdot o_C] = \mathbf{m}[\mathbf{fp} \cdot o_A] + \mathbf{m}[\mathbf{fp} \cdot o_B]$, the 192-bit field sum (in E).

- **Columns:** $\mathbf{pc}, \mathbf{fp}$; operands o_A, o_B, o_C; the two read values $(v_{A,i})_{i\in\{0,1,2\}}$ and $(v_{B,i})_{i\in\{0,1,2\}}$; memory counts r_A, r_B, r_C; bytecode count r_{bc}.
- **Constraints:** none.
- **Flushes:**
  - **State:** PULL $(\mathsf{ST}, \mathbf{pc}, \mathbf{fp})$, PUSH $(\mathsf{ST}, \mathsf{g} \cdot \mathbf{pc}, \mathbf{fp})$.
  - **Bytecode:** $\mathsf{read}\,(\mathsf{BC}, \mathbf{pc}, r_{\mathrm{bc}}, \mathsf{op}_{\mathsf{XOR}}, o_A, o_B, o_C, 0, 0)$.
  - **Memory:**

$$\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_X, r_X, v_{X,0}, v_{X,1}, v_{X,2}) \quad (X \in \{A, B\}),$$
$$\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_C, r_C, v_{A,0} + v_{B,0}, v_{A,1} + v_{B,1}, v_{A,2} + v_{B,2}).$$

7.2 MUL_NATIVE

MUL_NATIVE $[o_A, o_B, o_C]$ asserts $\mathbf{m}[\mathbf{fp} \cdot o_C] = \mathbf{m}[\mathbf{fp} \cdot o_A] \cdot \mathbf{m}[\mathbf{fp} \cdot o_B]$, the 192-bit field multiplication (in E).

- **Columns:** $\mathbf{pc}, \mathbf{fp}$; operands o_A, o_B, o_C; the two read values $(v_{A,i})_{i\in\{0,1,2\}}$ and $(v_{B,i})_{i\in\{0,1,2\}}$; memory counts r_A, r_B, r_C; bytecode count r_{bc}.
- **Constraints:** none.
- **Flushes:**
  - **State:** PULL $(\mathsf{ST}, \mathbf{pc}, \mathbf{fp})$, PUSH $(\mathsf{ST}, \mathsf{g} \cdot \mathbf{pc}, \mathbf{fp})$.
  - **Bytecode:** $\mathsf{read}\,(\mathsf{BC}, \mathbf{pc}, r_{\mathrm{bc}}, \mathsf{op}_{\mathsf{MUL}}, o_A, o_B, o_C, 0, 0)$.
  - **Memory** (the destination carries $v_A v_B$ in E, of degree two). With $y^3 = y + 1$, put

$$p_0 = v_{A,0}v_{B,0}, \quad p_1 = v_{A,0}v_{B,1} + v_{A,1}v_{B,0}, \quad p_2 = v_{A,0}v_{B,2} + v_{A,1}v_{B,1} + v_{A,2}v_{B,0},$$
$$p_3 = v_{A,1}v_{B,2} + v_{A,2}v_{B,1}, \qquad p_4 = v_{A,2}v_{B,2},$$

    twelve products over the nine pairs $v_{A,i}v_{B,j}$:

$$\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_X, r_X, v_{X,0}, v_{X,1}, v_{X,2}) \quad (X \in \{A, B\}),$$
$$\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_C, r_C, p_0 + p_3, p_1 + p_3 + p_4, p_2 + p_4).$$

7.3 SET_CONSTANT

SET_CONSTANT $[o, k_0, k_1, k_2]$ asserts $\mathbf{m}[\mathbf{fp} \cdot o] = k_0 + k_1 y + k_2 y^2$.

- **Columns:** $\mathbf{pc}, \mathbf{fp}$; operand o; immediate $(k_i)_{i\in\{0,1,2\}}$; memory count r; bytecode count r_{bc}.
- **Constraints:** none.
- **Flushes:**
  - **State:** PULL $(\mathsf{ST}, \mathbf{pc}, \mathbf{fp})$, PUSH $(\mathsf{ST}, \mathsf{g} \cdot \mathbf{pc}, \mathbf{fp})$.
  - **Bytecode:** $\mathsf{read}\,(\mathsf{BC}, \mathbf{pc}, r_{\mathrm{bc}}, \mathsf{op}_{\mathsf{SET}}, o, k_0, k_1, k_2, 0)$.
  - **Memory:** $\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o, r, k_0, k_1, k_2)$.

7.4 DEREF

DEREF $[o_1, o_2, o_3; \mathsf{mode}]$ reads a pointer and asserts the cell it points to. The row reads three cells:

- the *pointer* p, at $\mathbf{fp} \cdot o_1$;
- the *target* v_2, at the dereferenced address $p \cdot o_2$;
- the *local* value v_3, at $\mathbf{fp} \cdot o_3$.

The store asserts that v_2 equals the source chosen by the mode mode (§2): the local cell v_3, the return address $\mathsf{g}^2 \cdot \mathbf{pc}$, or the frame pointer $\mathbf{fp}$. The mode is two boolean flags $(f_{\mathbf{pc}}, f_{\mathbf{fp}})$: $(0,0)$ for `deref_cell`$[o_3]$, $(1,0)$ for `deref_pc`, $(0,1)$ for `deref_fp`.

- **Columns:** $\mathbf{pc}, \mathbf{fp}$; operands o_1, o_2, o_3; flags $f_{\mathbf{pc}}, f_{\mathbf{fp}}$; pointer p (a single K-limb); the local value $(v_{3,i})_{i\in\{0,1,2\}}$; memory counts r_1, r_2, r_3; bytecode count r_{bc}.
- **Constraints:** none.
- **Flushes:**
  - **State:** PULL $(\mathsf{ST}, \mathbf{pc}, \mathbf{fp})$, PUSH $(\mathsf{ST}, \mathsf{g} \cdot \mathbf{pc}, \mathbf{fp})$.
  - **Bytecode:** $\mathsf{read}\,(\mathsf{BC}, \mathbf{pc}, r_{\mathrm{bc}}, \mathsf{op}_{\mathsf{DRF}}, o_1, o_2, o_3, f_{\mathbf{pc}}, f_{\mathbf{fp}})$.
  - **Memory**, the target giving v_3, $\mathsf{g}^2 \cdot \mathbf{pc}$ or $\mathbf{fp}$ at the three flag settings:

$$\begin{aligned}
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_1, r_1, p, 0, 0),\\
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_3, r_3, v_{3,0}, v_{3,1}, v_{3,2}),\\
&\mathsf{read}\,(\mathsf{MEM}, p \cdot o_2, r_2, \bar{f}\, v_{3,0} + f_{\mathbf{pc}}\,(\mathsf{g}^2 \cdot \mathbf{pc}) + f_{\mathbf{fp}}\,\mathbf{fp},\ \bar{f}\, v_{3,1},\ \bar{f}\, v_{3,2}), \qquad \bar{f} = 1 + f_{\mathbf{pc}} + f_{\mathbf{fp}}.
\end{aligned}$$

7.5 JUMP

JUMP $[o_c, o_d, o_f]$ performs a conditional jump on whether the condition $v_{\mathrm{cond}} = \mathbf{m}[\mathbf{fp} \cdot o_c]$ is nonzero. It transfers to $\mathbf{pc} \leftarrow v_{\mathbf{pc}} = \mathbf{m}[\mathbf{fp} \cdot o_d]$, $\mathbf{fp} \leftarrow v_{\mathbf{fp}} = \mathbf{m}[\mathbf{fp} \cdot o_f]$ when $v_{\mathrm{cond}} \neq 0$, and defaults to $\mathbf{pc} \leftarrow \mathsf{g} \cdot \mathbf{pc}$, $\mathbf{fp} \leftarrow \mathbf{fp}$ otherwise. $v_{\mathrm{cond}}, v_{\mathbf{fp}}, v_{\mathbf{pc}}$ must be K-valued. The branch is taken according to a boolean indicator column $b = [\, v_{\mathrm{cond}} \neq 0\,]$, certified by a prover-supplied inverse w and 2 constraints:

- **Columns:** $\mathbf{pc}, \mathbf{fp}$; operands o_c, o_d, o_f; the condition v_{cond}, the destination $v_{\mathbf{pc}}$ and the frame $v_{\mathbf{fp}}$; inverse w (honest prover sets $w = v_{\mathrm{cond}}^{-1}$ if $v_{\mathrm{cond}} \neq 0$, otherwise $w = 0$); boolean indicator b; memory counts r_c, r_d, r_f; bytecode count r_{bc}.
- **Constraints:** $b = v_{\mathrm{cond}} \cdot w$ and $v_{\mathrm{cond}}\,(b+1) = 0$
- **Flushes:**
  - **State:** PULL $(\mathsf{ST}, \mathbf{pc}, \mathbf{fp})$, PUSH $(\mathsf{ST}, b\, v_{\mathbf{pc}} + b\,(\mathsf{g} \cdot \mathbf{pc}) + \mathsf{g} \cdot \mathbf{pc},\ b\, v_{\mathbf{fp}} + b\,\mathbf{fp} + \mathbf{fp})$.
  - **Bytecode:** $\mathsf{read}\,(\mathsf{BC}, \mathbf{pc}, r_{\mathrm{bc}}, \mathsf{op}_{\mathsf{JMP}}, o_c, o_d, o_f, 0, 0)$.
  - **Memory:**

$$\begin{aligned}
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_c, r_c, v_{\mathrm{cond}}, 0, 0),\\
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_d, r_d, v_{\mathbf{pc}}, 0, 0),\\
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_f, r_f, v_{\mathbf{fp}}, 0, 0).
\end{aligned}$$

7.6 BLAKE2S

BLAKE2S $[o_{m_0}, o_{m_1}, o_{m_2}, o_{m_3}, o_{cv}, o_{out}, o_{\mathrm{md}}]$ compresses a 64-byte message block under a 32-byte chaining value into a 32-byte result, with the BLAKE2s flags and byte counter in the metadata cell $\mathbf{fp} \cdot o_{\mathrm{md}}$. Each 128-bit half occupies one cell, top limb zero: the four message chunks at $\mathbf{fp} \cdot o_{m_0}, \ldots, \mathbf{fp} \cdot o_{m_3}$, addressed independently, the chaining value at the consecutive pair $\mathbf{fp} \cdot o_{cv}, \mathsf{g} \cdot \mathbf{fp} \cdot o_{cv}$, and the result at $\mathbf{fp} \cdot o_{out}, \mathsf{g} \cdot \mathbf{fp} \cdot o_{out}$.

Flock [BRW26] proves the compression relation with a batched R1CS over a Boolean witness (Annex C). The witness is packed into q_{flock} via Ring-Switching (Annex A), with 64 bits per K-element.

The message, result, chaining value and metadata are all committed in q_{flock}: the two low limbs z_0, z_1 of each of the nine cells $z \in \{m_0, \ldots, m_3, cv_0, cv_1, out_0, out_1, \mathrm{md}\}$, eighteen in all. Linking these values to the memory interaction implicitly forces the top limb of every memory cell accessed by BLAKE2s to be zero. The generalized R1CS matrices take the chaining value, byte counter, and the two finalization flags as free input: the memory interactions via the bus bind all four.

- **Columns:** $\mathbf{pc}, \mathbf{fp}$; operands $o_{m_0}, o_{m_1}, o_{m_2}, o_{m_3}, o_{cv}, o_{out}, o_{\mathrm{md}}$; one memory count r_z per cell z; bytecode count r_{bc}. The eighteen limbs are committed in q_{flock}, no duplicate columns needed. The nine addresses are the products $\mathsf{g}^k \mathbf{fp} \cdot o$, $k \in \{0, 1\}$.
- **Constraints:** none. The compression relation is handled by Flock.
- **Flushes:**
  - **State:** PULL $(\mathsf{ST}, \mathbf{pc}, \mathbf{fp})$, PUSH $(\mathsf{ST}, \mathsf{g} \cdot \mathbf{pc}, \mathbf{fp})$.
  - **Bytecode:** $\mathsf{read}\,(\mathsf{BC}, \mathbf{pc}, r_{\mathrm{bc}}, \mathsf{op}_{\mathsf{B2S}}, o_{m_0}, o_{m_1}, o_{m_2}, o_{m_3}, o_{cv}, o_{out}, o_{\mathrm{md}})$.
  - **Memory:**

$$
\begin{aligned}
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_{m_i}, r_{m_i}, m_{i,0}, m_{i,1}, 0) \quad (i = 0, \ldots, 3),\\
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_{cv}, r_{cv_0}, cv_{0,0}, cv_{0,1}, 0),\\
&\mathsf{read}\,(\mathsf{MEM}, \mathsf{g} \cdot \mathbf{fp} \cdot o_{cv}, r_{cv_1}, cv_{1,0}, cv_{1,1}, 0),\\
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_{out}, r_{out_0}, out_{0,0}, out_{0,1}, 0),\\
&\mathsf{read}\,(\mathsf{MEM}, \mathsf{g} \cdot \mathbf{fp} \cdot o_{out}, r_{out_1}, out_{1,0}, out_{1,1}, 0),\\
&\mathsf{read}\,(\mathsf{MEM}, \mathbf{fp} \cdot o_{\mathrm{md}}, r_{\mathrm{md}}, \mathrm{md}_0, \mathrm{md}_1, 0).
\end{aligned}
$$

8 End-to-end protocol

8.1 Bytecode encoding

The bytecode (of $N_{\mathrm{prog}} = 2^{\kappa_{\mathrm{bc}}}$ instructions) is encoded as one K-valued multilinear P in $\kappa_{\mathrm{bc}} + 4$ variables:

1. lay instruction z out as 16 slots of K: the opcode in slot 3, the operands and immediate lanes in slots $4, \ldots, 10$, and 0 everywhere else.
2. read the resulting $2^{\kappa_{\mathrm{bc}}} \times 16$ table as P, the low κ_{bc} variables indexing the instruction and the high four its slot, bits low first.

Instruction	3	4	5	6	7	8	9	10
XOR	$\mathsf{op}_{\mathsf{XOR}}$	o_A	o_B	o_C	0	0	0	0
MUL_NATIVE	$\mathsf{op}_{\mathsf{MUL}}$	o_A	o_B	o_C	0	0	0	0
SET_CONSTANT	$\mathsf{op}_{\mathsf{SET}}$	o	k_0	k_1	k_2	0	0	0
DEREF	$\mathsf{op}_{\mathsf{DRF}}$	o_1	o_2	o_3	$f_{\mathbf{pc}}$	$f_{\mathbf{fp}}$	0	0
JUMP	$\mathsf{op}_{\mathsf{JMP}}$	o_c	o_d	o_f	0	0	0	0
BLAKE2S	$\mathsf{op}_{\mathsf{B2S}}$	o_{m_0}	o_{m_1}	o_{m_2}	o_{m_3}	o_{cv}	o_{out}	o_{md}

So $P(z, 1, 1, 0, 0)$ is instruction z's opcode, $P(z, 0, 0, 1, 0)$ its first operand, and $P(z, 0, 0, 0, 0) = P(z, 0, 0, 1, 1) = 0$.

8.2 Public input

The public input is the first two memory cells $\mathbf{m}[\mathsf{g}^0], \mathbf{m}[\mathsf{g}^1]$, two 192-bit words (with top limb 0) both parties know. The verifier samples $r_m \in E$, the prover sends $c_0, c_1 \in E$ claiming $c_\ell = \widetilde{\mathbf{m}_\ell}(r_m, 0, \ldots, 0)$, and the verifier checks

$$c_0 + y c_1 \stackrel{?}{=} (1 + r_m)\, \mathbf{m}[\mathsf{g}^0] + r_m\, \mathbf{m}[\mathsf{g}^1].$$

The PCS binds $(c_0, c_1, 0)$ to the three memory limbs at $(r_m, 0, \ldots, 0)$. A mismatch gives a nonzero polynomial of degree at most one, so it passes with probability at most $1/|E|$ (Lemma 3.8).

8.3 Filling the tables

A table is proven over a power-of-two number of rows. The prover reaches one by running extra instructions, in execution cycles disjoint from the program's own run. Proposition 6.1 about bus balance splits the rows into two groups:

- one walk from $(\mathbf{pc}_0, \mathbf{fp}_0)$ to $(\mathbf{pc}_{\text{final}}, \mathbf{fp}_{\text{final}})$
- closed walks

The prover can thus leverage these *closed walks* to fill every table to its next power of 2. (important for completeness, but not for soundness)

8.4 Fiat Shamir instantiation

TODO

8.5 The unrolled protocol

Each reduction below adds evaluation claims on columns to a shared *claim pool*, proved by the final PCS Opening phase.

Setup. The Fiat-Shamir initial state (§8.4) is seeded by the public input, the bytecode and Flock's BLAKE2s R1CS matrices. The prover sends the memory log-size κ_{mem} and the six table log-heights τ_j; the verifier checks them against the caps (§6.2) and derives the layout of every stack (PCS and GKR leaves).

Commitment.

1. **Prover.** Stack (§4.1) every column, the three memory limbs, the two finalize (memory / bytecode) counts, and Flock's packed witness q_{flock} (§7.6), into q and sends its commitment (Annex B).
2. **Verifier.** Receives the commitment to q.

Bus.

1. **Verifier.** Samples and sends the fingerprint challenge $\alpha \in E^4$ and the multiset challenge $\beta \in E$ (§5.2).
2. **Prover.** Forms the push, pull, and *count* leaf vectors (§5.4; the count leaves are the count columns themselves, §6.2), stacked into blocks and padded with 1s. Sends one bus root R, then the count root R_c.
3. **Prover & Verifier.** Run the one batched GKR (§5.3) over all three trees. For the bus, the verifier checks each side's product against R; one R for both sides is the balance check (§5.2). For the counts it checks $R_c \neq 0$ (no read self-cancels, §6.2). The pass yields three leaf claims $\widetilde{V}_0^s(\zeta)$ at one shared ζ.
4. **Prover & Verifier.** Decompose each leaf claim (§5.4). The verifier forms every public part itself: the block selectors, the constant entries, the index column (§6.5), and the program's whole share of the two bytecode blocks, one evaluation at (ζ, α) (§8.1). Three blocks per side belong to no table: the state boundary, plus each array's (memory / bytecode) seed (push side) or finalization (pull side). For their committed columns the prover sends one evaluation each, entering the claim pool. Every other block belongs to a table, and is discharged by the table sumcheck (§5.5).

Table sumcheck. One run proves the six tables' constraints together with their share of the bus (§5.5).

1. **Verifier.** Samples and sends the batch challenge $\xi \in E$. Its first $N_{\text{cons}} = \sum_j N_{\text{cons},j}$ powers weight the constraints, a disjoint range per table; the last N_{side} weight the bus forms, one per side and shared by every table. It draws no point: the batch runs at the bus point, T_j tested at $\zeta_{<\tau_j}$.
2. **Prover & Verifier.** Run the sumcheck, $\tau_{\max} = \max_j \tau_j$ rounds with T_j joining at round $\tau_{\max} - \tau_j$, against the target the verifier derived from the three leaf claims. Its summand has individual degree three and the round check pins one of the four coefficients (§3), so a round costs three E-elements.
3. **Prover & Verifier.** The prover sends one value per column of every table. The verifier rebuilds each $C_{j,i}$ and B_j^s from them and checks that they reproduce the sumcheck's final value; the column values then enter the claim pool, T_j's at $\chi_{<\tau_j}$. `BLAKE2S`'s eighteen value limbs are columns of its table but live in q_{flock}, not the stack, so their claims route there (§7.6).

Public input.

1. **Verifier.** Sends $r_m \in E$ and forms the line $(1 + r_m)\mathbf{m}[\mathsf{g}^0] + r_m\mathbf{m}[\mathsf{g}^1]$ from the public words.
2. **Prover & Verifier.** The prover sends c_0, c_1; the verifier checks $c_0 + yc_1$ against the line and pools $(c_0, c_1, 0)$ as claims on $\mathbf{m}_0, \mathbf{m}_1, \mathbf{m}_2$ at $(r_m, 0, \ldots, 0)$ (§8.2).

BLAKE2s validity.

1. **Prover & Verifier.** Reduce the executed BLAKE2s constraints to one evaluation claim on q_{flock} (§7.6, Annex C).
2. **Prover & Verifier.** The ring switching of §4.2 reduces that bit-witness claim to a weighted-sum claim on the q_{flock} region of the stack, with an MLE-friendly E-valued weight (Definition 3.13); it joins the pool.

Opening.

1. **Prover & Verifier.** Each pooled claim is one evaluation of a column at a point, its value already supplied; via the stacking selectors (§4.1) it becomes a weighted sum $\sum_w W_j(w)\,\widetilde{q}(w) = c_j$ over the stacked witness. (The ring-switched claim arrives already in this form, its weight supported on the q_{flock} region.)
2. **Verifier.** Sends one batching challenge $\lambda \in E$, drawn after every claim value is bound. Claim j takes the weight λ^{j-1}, the ring-switched claim first and the pooled column claims after: the J weights are distinct powers of the one challenge (Theorem B.7).
3. **Prover & Verifier.** Fold the J claims into $W_\lambda = \sum_j \lambda^{j-1} W_j$ and $C_\lambda = \sum_j \lambda^{j-1} c_j$, and run the inner-product PCS opening on $\sum_w W_\lambda(w)\,\widetilde{q}(w) = C_\lambda$ (Annex B), the verifier evaluating W_λ itself. This one opening discharges every pooled claim; there is no separate reduction sumcheck.
4. **Verifier.** Accepts if every check above passed: the caps, the one bus root for both sides, the nonzero count root, every sumcheck's rounds and final value, the public-input line, flock's reduction, and the PCS opening.

9 ISA programming

How the zkDSL compiler (`crates/lean_compiler`; surface language in `crates/lean_compiler/zkDSL.md`) programs the six-instruction ISA. Two themes run through every pattern: *write-once memory is an assertion mechanism* (a second write of a cell is an equality check, an unwritten cell is prover-chosen), and *indices live in the exponent* (a logical index i is the element $\mathtt{g}^i$, so incrementing, offsetting, and address formation are single field multiplications).

9.1 Hints

The prover's nondeterminism enters through write-once cells that are still unset when execution needs them. Some are explicit advice (a fresh allocation pointer $\mathtt{g}^{\text{base}}$, a witness stream, a computed-advice builtin); others are back-solved by an instruction whose remaining operands are known. In every case the resulting word is unconstrained except by the instructions and memory equalities that consume it, so a program must check all security-relevant advice.

9.2 Division and inequality

The zkDSL's single-slash division $q = a/b$ uses write-once back-solving: it emits `MUL_NATIVE` with q as the one unset input and a as the already-written output. Witness generation fills $q = a\,b^{-1}$, while the ordinary multiplication table proves $qb = a$. It costs one VM instruction. Division by zero

is undefined. Double slash `//` remains compile-time integer floor division for sizes and indices; it is not a field operation.

The assertion `assert a != b` computes $x = a + b$ with `XOR`, takes a hinted $w = x^{-1}$, forms $p = x\,w$ with `MUL_NATIVE`, and writes 1 to p with `SET_CONSTANT`. Memory being write-once, the two writes to p agree only if $p = 1$, and $x = 0$ forces $p = 0$ whatever the hint.

9.3 Functions

A frame is fp-relative with layout [*retpc*, *retfp*, args, rets, locals]. A call: the callee frame pointer is hinted into a fresh cell *nfp*; the arguments and the caller's **fp** are stored through *nfp* with `DEREF`s (modes `cell` and `fp`); a `DEREF` in mode `pc` stores the return address $\mathsf{g}^2 \cdot \mathbf{pc}$ (the instruction after the transfer); one `JUMP` enters the callee. Return is a single `JUMP` to $\mathbf{m}[\mathbf{fp} \cdot 1]$ with frame pointer $\mathbf{m}[\mathbf{fp} \cdot \mathsf{g}]$. Cost: $n_{\text{args}} + n_{\text{rets}} + 4$ instructions per call.

The callee frame pointer being hinted, the following convention is crucial: *a function's internal logic should only depend on its inputs, not on where its frame happens to land*, so that the value of *nfp* does not impact its execution.

The ISA makes it possible to use values of **fp** outside the range $\{\mathsf{g}^0, \ldots, \mathsf{g}^{2^{\kappa_{\text{mem}}}-1}\}$. As long as every memory access of the form $\mathbf{m}[\mathbf{fp} \cdot o]$ (for some offset o) is in range, i.e. $\mathbf{fp} \cdot o = \mathsf{g}^i$ with $i < 2^{\kappa_{\text{mem}}}$, the execution is valid. Yet, reasoning about such out-of-range values of **fp** is tricky. The initial **fp** is in range, since the state boundary condition fixes $\mathbf{fp}_0 = \mathsf{g}^0$ (§6.1), but nothing guarantees it for the ones a `JUMP` sets. We thus suggest the following programming convention: whenever a `JUMP` updates **fp**, the program should guarantee that the new value is in range. The calling convention above does it for free:

- On a call, $\mathbf{fp} \leftarrow \mathbf{m}[\mathbf{fp} \cdot o_f]$, where $\mathbf{m}[\mathbf{fp} \cdot o_f] = \mathit{nfp}$ is hinted, hence arbitrary. But the callee frame was initialized with the `DEREF` $\mathbf{m}[\mathit{nfp}] \leftarrow \mathsf{g}^2 \cdot \mathbf{pc}$, whose memory access at address *nfp* forces $\mathit{nfp} = \mathsf{g}^i$ with $i < 2^{\kappa_{\text{mem}}}$ (Corollary 6.5).
- On a return, $\mathbf{fp} \leftarrow \mathbf{m}[\mathbf{fp} \cdot \mathsf{g}]$ reads back what the caller wrote there, i.e. its own **fp**, in range by induction.

9.4 Loops

A counted loop keeps its counter in the exponent: it starts at g^{lo}, advances by one multiplication $i \leftarrow \mathsf{g} \cdot i$, and exits on reaching g^{hi}. The body is lowered as a tail-recursive helper function whose exit test *is* the recursive call's `JUMP` condition (the `XOR` $i + \mathsf{g}^{\text{hi}}$ is nonzero exactly while the loop must continue), so an iteration costs one call plus two instructions, with no is-zero gadget. Loop state is threaded through captured arguments or a heap buffer.

9.5 Conditionals

`if/else` on a field equality is one `XOR` plus one conditional `JUMP`: the taken jump goes to whichever block the test should *not* fall into (= falls into *then*, $\neq$ into *else*), so no negation is ever computed. Intra-function jump targets are bytecode addresses $\mathsf{g}^{\text{entry}+i}$, backpatched at layout.

A taken `JUMP` reloads **fp** from a cell, and the ISA has no **fp**-read; a branching function therefore materializes its own **fp** once: a `DEREF` in mode `fp` writes **fp** into a fresh one-cell heap buffer, a `DEREF` in mode `cell` copies it back into the frame (2 instructions, amortized; in `main`, $\mathbf{fp} = \mathsf{g}^0 = 1$ is already a constant). Bindings made inside a branch are local to it; branches communicate through write-once cells: only one branch executes, so both may write the *same* cell.

9.6 Checking that a value is in the subfield K

To check that $\mathbf{m}[\mathbf{fp} \cdot o_1]$ and $\mathbf{m}[\mathbf{fp} \cdot o_2]$ are both in K, it suffices to emit `JUMP` $[o_0, o_1, o_2]$ with $\mathbf{m}[\mathbf{fp} \cdot o_0] = 0$. Indeed, the jump falls through, while `JUMP` unconditionally requires all three values it reads to be in K.

9.7 Range checks

The range check *in the exponent* proves $x \in \{\mathsf{g}^0, \ldots, \mathsf{g}^{k-1}\}$, i.e. $\log_{\mathsf{g}} x < k$, in 3 cycles:

1. `DEREF` through x: the dereferenced address $x \cdot \mathsf{g}^0$ rides the memory interaction, which balances only for the $2^{\kappa_{\text{mem}}}$ addresses $\mathsf{g}^0, \ldots, \mathsf{g}^{2^{\kappa_{\text{mem}}}-1}$ (§6.3): the bus itself proves $x = \mathsf{g}^e$ with $e < 2^{\kappa_{\text{mem}}}$;
2. `MUL` $x \cdot y$ into a write-once cell holding the constant g^{k-1}: the runner back-solves the complement $y = \mathsf{g}^{k-1-e}$ (§9.1), and the double write asserts $x \cdot y = \mathsf{g}^{k-1}$;
3. `DEREF` through y: proves $y = \mathsf{g}^f$, $f < 2^{\kappa_{\text{mem}}}$.

Then $e + f \equiv k - 1 \pmod{2^{64} - 1}$ with $e, f < 2^{\kappa_{\text{mem}}} \leq 2^{32}$, so $e + f < 2^{33} < 2^{64} - 1$ and the congruence is an equality over the integers; a "negative" $k - 1 - e$ would wrap to $\approx 2^{64} \gg 2^{\kappa_{\text{mem}}}$, so $e \leq k - 1$, for every memory size the prover may announce, provided $k \leq 2^{16}$ (the minimum, §2). The two `DEREF` targets are deferred touches; the constant cell is one amortized `SET` per distinct bound per frame.

9.8 Counter in BLAKE2s

Hashing a slice of constant size with BLAKE2s is straightforward, by hardcoding the counter inside each md (using the `SET_CONSTANT` instruction).

Hashing slice of dynamic-size requires to reconstruct the counter at runtime. One efficient way to handle this is the following: absorb a window of $B = 2^b$ blocks per loop iteration, unrolled, so that block j of window q has counter $64Bq + 64(j + 1)$. The two terms have disjoint bits, the base $64Bq = 2^{6+b}q$ having none below $6 + b$, and K is a polynomial basis, so a plain `XOR` adds them as integers: each md is one instruction against a compile-time constant, $\mathit{base} + (64(j + 1) \,\|\, f_0 \,\|\, f_1)$. Only a window's last block overlaps, and it takes the next window's base instead. That base is the loop counter g^q decomposed into hinted bits, tied back by $\prod_j (\mathsf{g}^{2^j})^{b_j} = \mathsf{g}^q$ and summed weighted by 2^{6+b+j}: one decomposition per window, so under three instructions a block at $B = 32$.

9.9 Match statements

Matching on $x = \mathsf{g}^j$, $j \in [0, n)$, the dispatch jumps to

$$d \;=\; \mathsf{g}^T \cdot x^2 \;=\; \mathsf{g}^{T+2j},$$

slot j of a *trampoline* table at bytecode base T, where a slot is `SET` $c = \mathsf{g}^{\text{block}_j}$ then `JUMP` c. The case blocks sit anywhere, unaligned and of any length, the only arithmetic being the squaring x^2. Two jumps per dispatch, ≈ 7 cycles independent of n.

The slots are two instructions rather than one because a `JUMP` reads its destination from a *cell*: a one-instruction slot would need its target cell pre-written, i.e. n `SET`s executed before every dispatch. (Other layouts exist and trade exactly this, e.g. a memory-resident table of the n block addresses,

read with one DEREF and taken with a *single* jump, paying the SETs as setup; worthwhile for many repeated small matches. The compiler currently emits only the trampoline.)

The matched value must be known to lie in $[0, n)$: a hinted x must be range-checked first (§9.7); otherwise the dispatch lands at an attacker-chosen program counter.

10 Ethereum use cases

An aggregate proof certifies signature claims and the correct encoding of blob data. One guest program establishes these claims by checking raw signatures, checking a blob matrix, or recursively verifying child proofs of the same program.

10.1 Recursive aggregation

A signature claim identifies what was signed and by whom. For XMSS [HBG+18], it is a public key, an epoch, and a message; each epoch has one message across the aggregate, and its keys form one group. For SPHINCS+ [BHK+19], it is a (key, message) pair.

A node may publish any subset of the signature claims supported by its raw signatures and verified children. The default is their sorted, deduplicated union. The essential condition is coverage: every published claim must be justified, either directly at this node or recursively through a child.

The DA claim is a sorted, duplicate-free list of at most 16 LeanDA roots, possibly empty. Each root is paired with the hash of its membership test vector L (§10.2.2).

10.1.1 The public input

The proof carries the signature claims and DA root list. Their digests bind these variable-length lists into a fixed-size statement. The VM's public input is BLAKE2s of the following fields, in order:

32 bytes	Fiat-Shamir IV
32 bytes	signer-set digest
32 bytes	DA list digest
$(\kappa_{\mathrm{bc}} + 5) \times 24$ bytes	deferred bytecode point and value
30×24 bytes	deferred matrix point and two values

The deferred claims let recursion postpone expensive polynomial evaluations until the final verification (§10.1.3).

The DA list digest is BLAKE2s of the concatenated pairs $(\mathsf{root}, \mathsf{h}_L)$, each a 32-byte DA root root followed by a 32-byte hash h_L of the membership test vector L.

The signer-set digest is BLAKE2s of the following, where each cell is 16 bytes and a count is encoded as $\mathsf{g}^{\mathrm{count}}$ (8 bytes, zero-padded to one cell):

block 0	XMSS group count, SPHINCS+ claim count, SPHINCS+ list digest
per XMSS group	epoch, key count, 32-byte message
	key list digest, two zero cells

Each inner list digest is BLAKE2s of the list alone: 32 bytes per XMSS key or 64 bytes per SPHINCS+ (key, message) pair.

10.1.2 Checking coverage

Every published claim needs support from a direct check or a recursively verified child. The guest builds a table with a region per XMSS (epoch, message) group and separate regions for SPHINCS+ and DA. Each region starts with the claims the parent publishes, followed by extra slots for duplicates and omitted claims.

Each occurrence of a supporting claim needs its own slot. If two children prove the same published claim, one occurrence uses its published slot and the other uses an extra slot. Every occurrence of a claim the parent omits uses an extra slot. These claims are still verified; only the published prefixes enter the parent's list digests.

The prover supplies the extra claims and assigns each supporting occurrence to a slot. The guest checks that they match, confines the assignment to the correct region, and writes a distinct counter value there. Write-once memory prevents slot reuse. Requiring as many writes as slots ensures that every slot is covered, including every published claim.

10.1.3 Deferred evaluation claims

Verifying a child requires evaluations of three fixed polynomials: the bytecode multilinear, in $\kappa_{\mathrm{bc}} + 4$ variables (§8.1), and flock's R1CS matrix multilinears $\widetilde{A}_0, \widetilde{B}_0$, each in 28 variables (Annex C). Computing them inside the guest would be very expensive. Instead, the guest carries evaluation claims in its statement. These expensive computations are moved 'outside of the snark'.

A node combines the claims carried by its children with the new claims produced while verifying them. Two sumchecks reduce these to three evaluations: one of the bytecode polynomial and one of each matrix polynomial, batched. The Fiat-Shamir for batching and sumchecks is seeded by all child and running claims.

10.2 Data availability: LeanDA

Erasure coding makes data recoverable from a subset of its encoded symbols. Sampling checks whether those symbols can be retrieved, but does not establish that they form a valid encoding. An approach using leanVM was introduced in Risitano's presentation and developed in LeanDA [MWKR26] by Meng, Wagner, Kadianakis and Risitano: commit to an encoded matrix and prove every row is a codeword, then sample authenticated cells. A formal security proof can be found in [Wag26].

Our construction uses BLAKE2s and the additive Reed-Solomon encoder over K already used by the PCS, with a different codeword membership test from LeanDA's (§10.2.2). The polynomial basis, transform, and membership lemmas are collected in Annex D.

10.2.1 The commitment

A blob is a polynomial in $K[X]$ of degree less than k, represented by its k evaluations on $\mathcal{U}_{\log_2 k}$. Its Reed-Solomon encoding is the vector of evaluations on the larger domain $\mathcal{D} = \mathcal{U}_{\log_2 n}$, where $n = 2k$. The first k symbols are the original blob, so the encoding is systematic; the remaining symbols provide redundancy. Any k encoded evaluations recover the blob.

One commitment covers a nonempty matrix of N_{blob} encoded blobs, one per row.

Split each encoded row into cells of N_{cell} symbols. The guest fixes these dimensions, matching the byte sizes of an EIP-4844 blob and an EIP-7594 sampling cell:

k	2^{14} symbols	128 KiB of payload per row
n	2^{15} symbols	256 KiB per encoded row
N_{cell}	2^8 symbols	2 KiB per cell
n/N_{cell}	128 cells	any 64 reconstruct a row
N_{blob}	1 to 1024	dynamic parameter

Zero codewords pad the row count to the next power of two. The root binds this padded matrix.

Let H denote BLAKE2s and $e_{i,j}$ the hash of cell j in row i. The same cell digests feed two branches:

- **Row branch.** Hash the first k/N_{cell} cell digests of each row into r_i, then Merkle-commit these row digests into $\mathsf{root}_{\text{row}}$.
- **Column branch.** Merkle-commit each column of cell digests into C_j, then Merkle-commit the column roots into $\mathsf{root}_{\text{col}}$.

The matrix root is $\mathsf{root} = H(\mathsf{root}_{\text{row}}, \mathsf{root}_{\text{col}})$ (Figure 1). See [MWKR26] for details.

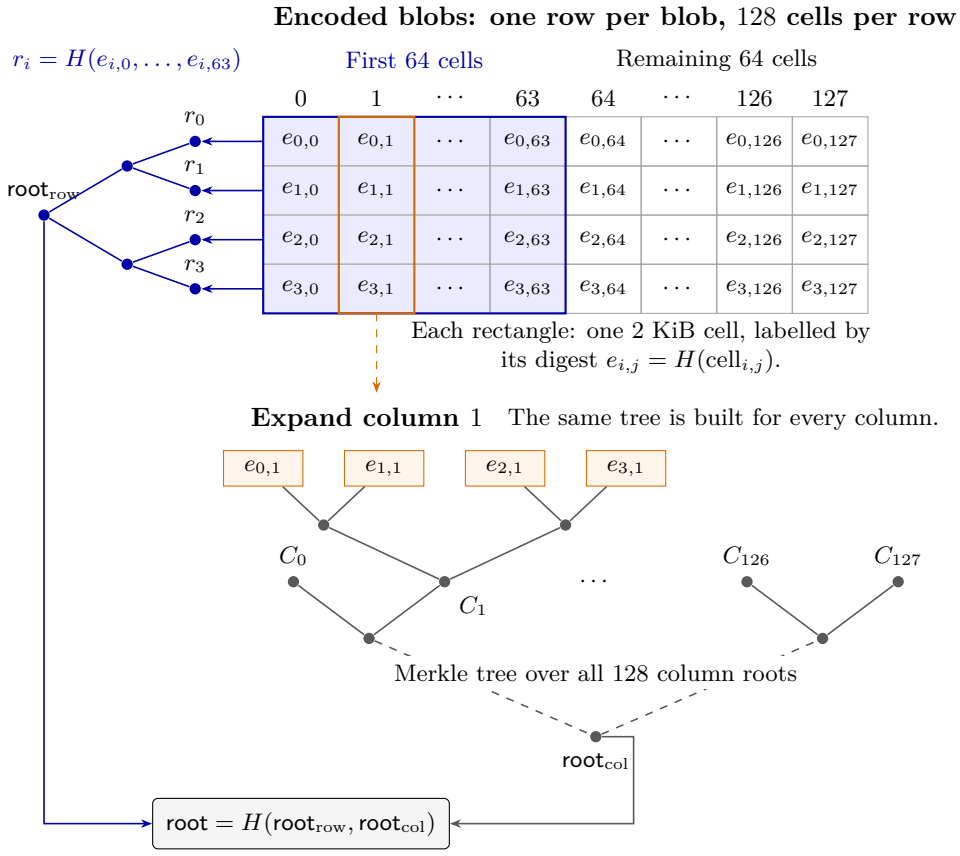


Figure 1: Example of encoding of 4 blobs.

10.2.2 Codeword membership

A row is valid exactly when it consists of evaluations of a polynomial of degree less than k. On our additive domain at rate 1/2, such rows are orthogonal to every codeword, and this condition also characterizes them: the code is self-dual (Corollary D.9). This gives an efficient membership test.

The Fiat-Shamir challenges $\log_2 k$ in E are derived from the matrix root root. These define the polynomial L of (26) and its evaluation vector $\mathbf{L} = (L(x))_{x\in\mathcal{D}}$. Let $\mathsf{h}_L = H(\mathbf{L})$, serializing each entry as its three little-endian 64-bit limbs, in domain order.

The guest recomputes root from the encoded rows, receives $\mathbf{L}$ as a hinted vector, checks its hash against h_L, and uses it to check membership:

$$\langle \mathbf{L},\, w_i\rangle = \sum_{x\in\mathcal{D}} L(x)w_i(x) \;\stackrel{?}{=}\; 0.$$

for every row w_i. The vector is shared across all rows and hashed once. It contains 32768 entries, or 768 KiB.

Recursive nodes preserve the pair $(\mathsf{root}, \mathsf{h}_L)$ for each retained claim. The final native verifier recomputes h_L from every retained root (outside of the snark).

Every valid row passes. Under uniform challenges, a fixed invalid matrix passes with probability at most $\log_2 k/|E| = 14/2^{192}$ (Proposition D.10).

A Ring switching

Ring switching was introduced in [DP26]. We present a variant tailored to our use case: transforming a PCS for K into a PCS for $\mathbb{F}_2$ (transforming a K-valued multilinear PCS, with κ_{flock} variables and opening claims in E, into a commitment protocol to 64 $\mathbb{F}_2$-valued multilinears, each with κ_{flock} variables and sharing opening claims in E).

A.1 Setting

The two fields are those of §2: $K = \mathbb{F}_2[x]/(x^{64}+x^4+x^3+x+1)$ inside $E = K[y]/(y^3+y+1)$. As $\mathbb{F}_2$-vector spaces, K has basis $x^0,\dots,x^{63}$, and E has the 192 products

$$e_w = x^i y^j \in E, \qquad w = 64j+i, \quad 0\le i<64, \quad 0\le j<3.$$

The witness is 64 bit-valued multilinears $Q_0,\dots,Q_{63} : \{0,1\}^{\kappa_{\text{flock}}} \to \{0,1\}$. We commit to them packed, one K element per point:

$$q_{\text{flock}} : \{0,1\}^{\kappa_{\text{flock}}} \to K, \qquad q_{\text{flock}}(u) = \sum_{i=0}^{63} Q_i(u)\, x^i.$$

The commitment scheme proves statements of one shape. Given a target $T \in E$ and an MLE-friendly weight $W : \{0,1\}^{\kappa_{\text{flock}}} \to E$ (Definition 3.13), an opening proves

$$\sum_{u\in\{0,1\}^{\kappa_{\text{flock}}}} W(u)\, q_{\text{flock}}(u) = T.$$

The weight W and target T are E-valued, the committed values are K-valued.

What arrives is not of that shape. It is 64 claimed values $s_i \in E$, one per bit polynomial, at a point $r \in E^{\kappa_{\text{flock}}}$ they all share.

$$s_i \stackrel{?}{=} \widetilde{Q_i}(r) = \sum_{u\in\{0,1\}^{\kappa_{\text{flock}}}} \mathrm{eq}(r,u)\, Q_i(u), \qquad 0\le i<64,$$

Ring switching [DP26] turns those 64 claims into a single statement the opening can prove.

A.2 The reduction

Fix an $\mathbb{F}_2$-linear map $\Phi : E \to E$:

$$\begin{aligned}\sum_{u\in\{0,1\}^{\kappa_{\text{flock}}}} \Phi\big(\mathrm{eq}(r,u)\big)\, q_{\text{flock}}(u) &= \sum_{i=0}^{63} x^i \sum_{u\in\{0,1\}^{\kappa_{\text{flock}}}} Q_i(u)\, \Phi\big(\mathrm{eq}(r,u)\big) \\ &= \sum_{i=0}^{63} x^i\, \Phi\Big(\sum_{u\in\{0,1\}^{\kappa_{\text{flock}}}} Q_i(u)\ \mathrm{eq}(r,u)\Big) = \sum_{i=0}^{63} x^i\, \Phi(s_i).\end{aligned} \tag{5}$$

The reduction is:

1. once the s_i are fixed, the verifier draws Φ at random;
2. it computes the target $T = \sum_{i=0}^{63} x^i \Phi(s_i)$;
3. the commitment scheme opens q_{flock} against the weight $W : \{0,1\}^{\kappa_{\text{flock}}} \to E$, $W(u) = \Phi\big(\mathrm{eq}(r,u)\big)$, with target T; if that opening accepts, the verifier accepts the 64 claims.

Everything now turns on which Φ we can afford to draw.

A.3 Which maps are sound

The error, transposed. Suppose the s_i are wrong. The errors $\delta_i = s_i - \widetilde{Q_i}(r) \in E$, for $0 \le i < 64$, are not all zero, and they are fixed before Φ is drawn. By (5) the target the verifier computes differs from the honest one by $\sum_{i=0}^{63} x^i \Phi(\delta_i)$, so the opening succeeds only if

$$\sum_{i=0}^{63} x^i\, \Phi(\delta_i) = 0.$$

Expand each error in the $\mathbb{F}_2$-basis of E, say $\delta_i = \sum_{w=0}^{191} \delta_{i,w} e_w$ with $\delta_{i,w} \in \mathbb{F}_2$, and exchange the two sums:

$$\sum_{i=0}^{63} x^i\, \Phi(\delta_i) = \sum_{i=0}^{63} x^i \sum_{w=0}^{191} \delta_{i,w}\, \Phi(e_w) = \sum_{w=0}^{191} \Phi(e_w)\, \Delta_w, \qquad \Delta_w := \sum_{i=0}^{63} \delta_{i,w} x^i \in K. \tag{6}$$

The x^i are an $\mathbb{F}_2$-basis of K, so some $\Delta_w \neq 0$. The cheating prover succeeds if $\sum_{w=0}^{191} \Phi(e_w)\, \Delta_w = 0$. We must thus draw Φ to make this event unlikely.

Linear maps are Frobenius sums. Squaring is additive in characteristic 2, so

$$\Phi(a) = \sum_{k=0}^{191} c_k\, a^{2^k}, \qquad a \in E, \quad c_k \in E,$$

is $\mathbb{F}_2$-linear for any choice of coefficients. Every $\mathbb{F}_2$-linear map $E \to E$ arises this way, and exactly once. Distinct tuples of (c_k) give distinct maps: their difference would be a nonzero polynomial of degree at most 2^{191} with 2^{192} roots. There are $|E|^{192}$ tuples and as many maps. Call $\mathcal{S} = \{k : c_k \neq 0\} \subseteq \{0, \ldots, 191\}$ the *support* of Φ. Substituting into the check above,

$$\sum_{w=0}^{191} \Phi(e_w)\, \Delta_w = \sum_{w=0}^{191} \sum_{k\in\mathcal{S}} c_k\, e_w^{2^k}\, \Delta_w = \sum_{k\in\mathcal{S}} c_k\, \Omega_k, \qquad \Omega_k := \sum_{w=0}^{191} e_w^{2^k}\, \Delta_w \in E. \tag{7}$$

The Ω_k depend only on the error, the c_k only on the randomness. So we need the map $K^{192} \to E^{|\mathcal{S}|}$, $\Delta \mapsto (\Omega_k)_{k\in\mathcal{S}}$, to be injective: a nonzero error has to leave some $\Omega_k \neq 0$ for the coefficients to act on.

Sixty-four terms. That map is K-linear, since the Δ_w enter linearly and the Frobenius touches only the constants e_w. It sends K^{192}, of K-dimension 192, into $E^{|\mathcal{S}|}$, of K-dimension $3|\mathcal{S}|$, so injectivity needs $|\mathcal{S}| \geq 64$. Take $\mathcal{S} = \{0, \ldots, 63\}$ and suppose $\Omega_k = 0$ for all $k < 64$. Two steps force $\Delta = 0$.

First, split Ω_k over K. Write $w = 64j + i$, so that $e_w^{2^k} = x^{i2^k} \left(y^{2^k}\right)^j$ and

$$\Omega_k = \sum_{j=0}^{2} (y^{2^k})^j \, \Omega_{j,k}, \qquad \Omega_{j,k} := \sum_{i=0}^{63} \Delta_{64j+i} \, x^{i2^k} \in K, \qquad 0 \leq j < 3.$$

Squaring is a bijection of K onto itself, so $y^{2^k} \in K$ would force $y \in K$, hence $E = K[y] = K$. By Lemma 3.7 applied to y^{2^k}, the set $\{1, y^{2^k}, (y^{2^k})^2\}$ is a K-basis of E, and $\Omega_k = 0$ forces $\Omega_{0,k} = \Omega_{1,k} = \Omega_{2,k} = 0$.

Second, read the 64 equations $\Omega_{j,k} = 0$, at fixed j, as a single polynomial. Put

$$D_j(v) = \sum_{i=0}^{63} \Delta_{64j+i} \, v^i \in K[v], \qquad \text{so that} \qquad \Omega_{j,k} = D_j(x^{2^k}).$$

Then D_j has degree at most 63 and vanishes at the 64 points $x, x^2, x^4, \ldots, x^{2^{63}}$. Those are distinct, because x has degree 64 over $\mathbb{F}_2$ and so has 64 distinct conjugates. A nonzero polynomial of degree at most 63 cannot have 64 roots, so $D_j = 0$ and every Δ_{64j+i} vanishes. A nonzero error therefore leaves some $\Omega_k \neq 0$.

The count is tight on both sides: 64 terms give 64 equations $\Omega_k = 0$ over E, which is 192 equations over K, against exactly 192 unknowns $\Delta_w \in K$.

A.4 The map we draw

We need a random Φ of support $\{0, \ldots, 63\}$, cheap to apply. Instead of drawing its 64 coefficients independently, we derive them from six challenges.

Draw $f_0, \ldots, f_5 \leftarrow E$ once the s_i are fixed, and compose six two-term maps: for an input $a \in E$, put $a_0 = a$ and

$$a_{p+1} = a_p + f_p \, a_p^{2^{2^{5-p}}} \in E \quad (0 \leq p < 6), \qquad \Phi(a) = a_6. \tag{8}$$

Each stage is $\mathbb{F}_2$-linear, so Φ is. Write $k = \sum_{p=0}^{5} k_p \, 2^{5-p}$ for the binary digits of k. The composition expands to

$$\Phi(a) = \sum_{k=0}^{63} c_k \, a^{2^k}, \qquad c_k = \prod_{p \,:\, k_p = 1} f_p^{2^{(k \bmod 2^{5-p})}}. \tag{9}$$

Stage p fires when $k_p = 1$ and contributes a factor f_p, which every later stage that fires raises to its own Frobenius power; those shifts sum to k mod 2^{5-p}. Each k from 0 to 63 occurs exactly once, so the c_k are 64 distinct monomials in $f_0, \ldots, f_5$, one for every Frobenius power the previous section requires.

Soundness. Fix a nonzero error before the challenges are drawn. The previous section gives some $\Omega_k \neq 0$, and distinct monomials cannot cancel, so

$$\sum_{k=0}^{63} c_k \, \Omega_k$$

is a nonzero polynomial in $f_0, \dots, f_5$. Its total degree is at most the largest degree of a c_k, reached at $k = 63$ where every bit is set:

$$2^{31} + 2^{15} + 2^7 + 2^3 + 2 + 1 < 2^{32},$$

so by Schwartz–Zippel (Lemma 3.8) the check passes with probability less than $2^{32}/2^{192} = 2^{-160}$.

A.5 Evaluating the weight

An opening reduces $\sum_{u \in \{0,1\}^{\kappa_{\text{flock}}}} W(u)\, q_{\text{flock}}(u) = T$ to a claim about q_{flock} at a random point $r' \in E^{\kappa_{\text{flock}}}$, and the verifier must evaluate the multilinear extension of W at r', written $\widetilde{W}(r')$, by itself. One fact turns that into a short formula.

Fact (Frobenius passes through eq *on the boolean hypercube).* Squaring respects sums and products and fixes bits, so for $u \in \{0,1\}^{\kappa_{\text{flock}}}$, by Definition 3.4,

$$\operatorname{eq}(r, u)^{2^k} = \prod_{n=1}^{\kappa_{\text{flock}}} (1 + r_n + u_n)^{2^k} = \prod_{n=1}^{\kappa_{\text{flock}}} (1 + r_n^{2^k} + u_n) = \operatorname{eq}(r^{2^k}, u), \qquad r^{2^k} := (r_1^{2^k}, \dots, r_{\kappa_{\text{flock}}}^{2^k}).$$

Expand W at r' with (9) and this fact:

$$\begin{aligned} \widetilde{W}(r') &= \sum_{u \in \{0,1\}^{\kappa_{\text{flock}}}} W(u)\, \operatorname{eq}(r', u) = \sum_{u \in \{0,1\}^{\kappa_{\text{flock}}}} \Phi\big(\operatorname{eq}(r, u)\big)\, \operatorname{eq}(r', u) \\ &= \sum_{k=0}^{63} c_k \sum_{u \in \{0,1\}^{\kappa_{\text{flock}}}} \operatorname{eq}(r^{2^k}, u)\, \operatorname{eq}(r', u). \end{aligned}$$

The inner sum is $\operatorname{eq}(r^{2^k}, r')$, since $\operatorname{eq}(u, \cdot\,)$ is its own multilinear extension for every u. By Definition 3.4 again,

$$\boxed{\widetilde{W}(r') \;=\; \sum_{k=0}^{63} c_k \prod_{n=1}^{\kappa_{\text{flock}}} (1 + r_n^{2^k} + r'_n).}$$

A.6 Cost

One application of Φ takes 63 squarings and 6 multiplications: the Frobenius ladder climbs $32 + 16 + 8 + 4 + 2 + 1$ steps, and each stage multiplies once by its challenge. The same recurrence run on the coefficient table gives the c_k of (9): stage p raises its 2^p entries to the $2^{2^5 - p}$ power and multiplies each by f_p; the raising is 32 squarings whatever p is.

	squarings	multiplications
one application of Φ	63	6
the target T, 64 applications	4032	384
the coefficients c_k, once	192	63
the weight $\widetilde{W}(r')$	$63\kappa_{\text{flock}}$	$64\kappa_{\text{flock}}$

Everything above is in E. The 64 products $x^i\, \Phi(s_i)$ in the target are left out: x^i lies in K, so they are cheaper, and stepping x^i to x^{i+1} is a shift.

A.7 Credits

The construction above rests on generalizing ring switching [DP26] to extension degrees that are not powers of two, an idea of Lev Soukhanov, *[[alloc]init].* The map Φ follows [BRW26].

B The polynomial commitment scheme: WHIR /Ligerito

A self-contained description of the WHIR/Ligerito polynomial commitment scheme [ACFY25, NA25] over binary fields, with a round-by-round soundness analysis in the Johnson list-decoding regime. The two schemes coincide when WHIR is instantiated with interleaved codes and Ligerito with the Reed-Solomon codes of Annex D, which gives the novel polynomial basis and additive NTT used for encoding. Following Flock [BRW26], we do not use out-of-domain (OOD) sampling at commitment, so our PCS is only list binding. The two fields are those of §2: we commit to a K-valued multilinear, and opening claims are in E.

Symbols.

here	in [ACFY25, NA25, BSCH+25]	meaning
M_i	m_i	variables of the multilinear at level i
Λ	r	number of levels
ℓ_i	ℓ_i, ν	fold factor, and the interleaving exponent
ρ_i	ρ_i	code rate
ν_i	R_i	rate exponent, $\rho_i = 2^{-\nu_i}$
γ_i	γ_i	proximity radius
η_i	η_i	Johnson slack
θ	θ	MCA parameter (denoted m in [BSCH+25])
$\delta_{\min}$	δ	relative minimum distance
λ	λ, α	batching challenge, as in §4.1
χ_j	s_j	sumcheck fold challenge
$c^{(j)}$	σ_j	running sumcheck claim after round j
J_i	T_i	number of claims, as in §8.5
ω_i	t_i	query count
$\mathcal{Q}_i$	J_i	the sampled columns
n	n	code block length
$\mathcal{D}$	S	evaluation domain
$Z^{(i)}$	$C^{(i)}$	oracle, an interleaved word (C_j is a constraint)
N_{rows}	(none)	rows of $Z^{(0)}$ the prover commits, $N_{\text{rows}} \le 2^{\ell_0}$
N_{stack}	(none)	field elements placed in the stack, as in §4.1
$\mathcal{L}_\gamma$	λ_γ	list of codewords within γ
W, c	w, v	weight and value of a claim, as in §4.1
f^*	p	final plaintext multilinear
c_{ood}	y	out-of-domain answer (y generates E over K)
A_{mca}	a	numerator of the MCA error
$\mathcal{V}_i$, $\widehat{\mathcal{V}}_i$	W_i, $\widehat{W}_i$	subspace vanishing polynomial, normalized
$\mathcal{X}_j$	X_j	novel-basis polynomial (X_i is a sumcheck variable)
ψ_d	q_d	linearized map of the NTT (q is the witness stack)
B	B	butterfly array
$\mathcal{P}$	P	message polynomial (P_i is a committed column)
e_c	β_c	the fixed $\mathbb{F}_2$-basis, as in Annex A

B.1 Preliminaries

B.1.1 Codes, interleaving, and list decoding

We use the Reed-Solomon code $\mathrm{RS}[E, \mathcal{D}, k]$ of Definition D.2, with $|\mathcal{D}| = n$, dimension $k = 2^\kappa$, and rate $\rho = k/n$. Messages are indexed by $\{0,1\}^\kappa$; the linear encoder Enc of §B.2 maps them onto the code and preserves K-valued words.

Definition B.1 (Interleaved words, lists, folds)**.** An *interleaved word* is a matrix $Z \in E^{\{0,1\}^\ell \times \mathcal{D}}$; it is an *interleaved codeword* if every row $Z[u,\cdot]$ is a codeword ($u \in \{0,1\}^\ell$). Z *agrees* with an interleaved codeword U on $\mathcal{A} \subseteq \mathcal{D}$ if $Z[u,x] = U[u,x]$ for all $u \in \{0,1\}^\ell$ and all $x \in \mathcal{A}$; it is γ-*close* to U if they agree on some $\mathcal{A}$ with $|\mathcal{A}| \geq (1-\gamma)n$; and $\mathcal{L}_\gamma(Z)$ is the set of interleaved codewords γ-close to Z. A single word *agrees with the code* on $\mathcal{A}$ if some codeword equals it there, and Z *agrees with the interleaved code* on $\mathcal{A}$ if some interleaved codeword equals its rows there. An interleaved codeword U *decodes* to the multilinear $g_U : \{0,1\}^{\ell+\kappa} \to E$ with $U[u,\cdot] = \mathrm{Enc}(g_U(u,\cdot))$; distinct U decode to distinct g_U, hence distinct $\widetilde{g_U}$. For $\chi \in E^j$, $j \leq \ell$, the *fold* $Z_\chi \in E^{\{0,1\}^{\ell-j}\times\mathcal{D}}$ is $Z_\chi[u,x] := \sum_{\iota\in\{0,1\}^j} \mathrm{eq}(\chi,\iota)\, Z[(\iota,u),x]$ (for $j=\ell$, a single word in $E^{\mathcal{D}}$).

Fact B.2 (Folding commutes with decoding)**.** *If U is an interleaved codeword then so is U_χ, with $\widetilde{g_{U_\chi}} = \widetilde{g_U}(\chi,\cdot)$ (linearity of* Enc *plus Fact 3.6).*

Theorem B.3 (Johnson bound)**.** *For every interleaved word Z of* $\mathrm{RS}[E,\mathcal{D},2^\kappa]$ *(rate ρ) and every $\eta \in (0, 1-\sqrt{\rho})$: $|\mathcal{L}_{1-\sqrt{\rho}-\eta}(Z)| \leq \frac{1}{2\eta\sqrt{\rho}}$. (see §B.6 for a self-contained proof)*

B.1.2 PCS and round-by-round soundness

Definition B.4 (L-list binding, informal)**.** A *polynomial commitment scheme* (PCS) is *L-list binding* if every commitment transcript cm determines a set $\mathcal{G}_{\mathsf{cm}}$ of at most L multilinear polynomials, not necessarily efficiently computable, such that for every unbounded prover and every collection of opened claims: if no member of $\mathcal{G}_{\mathsf{cm}}$ satisfies all the claims, the verifier rejects except with small probability. $L = 1$ is standard binding.

Definition B.5 (RBR soundness, informal; cf. [CCH+19, ACFY25])**.** A public-coin protocol is *round-by-round (RBR) sound with errors* $(\varepsilon_1, \varepsilon_2, \dots)$, one per verifier message, if there is an *invariant* on partial transcripts (not necessarily efficiently computable) marking the positions where a cheating prover has already lost:

(i) on a false statement, the invariant holds before any message is sent;

(ii) the prover alone cannot escape it: it survives every prover message;

(iii) the i-th verifier message lets it escape with probability at most ε_i;

(iv) if it survives to the end of the interaction, the verifier rejects.

Interactive soundness error is then at most $\sum_i \varepsilon_i$. After Fiat–Shamir, it is $\max_i \varepsilon_i$ per random-oracle query [CCH+19, BSCS16].

B.2 Reed–Solomon codes over binary fields

The encoder $\mathrm{Enc}_{\kappa,\nu}$ of Definition D.3 evaluates 2^κ novel-basis coefficients on an additive domain of size $n = 2^{\kappa+\nu}$. Here $\nu \geq 1$ and $\kappa + \nu \leq 64$. The protocol uses two properties, proved in Annex D:

- *fast encoding, for the prover:* $\mathrm{Enc}_{\kappa,\nu}$ is computed by the additive NTT with $\frac{1}{2}\kappa n$ multiplications and fewer than $(\kappa+1)\,n$ additions (Proposition D.6);
- *MLE-friendly columns, for the verifier:* for every $x \in \mathcal{D}$, the weight $W_x : \{0,1\}^\kappa \to K$, defined by $\mathrm{Enc}_{\kappa,\nu}(g)[x] = \langle W_x, g\rangle$ (the relation being valid for arbitrary g), admits an MLE efficiently evaluable anywhere in E^κ in $O(\kappa)$ field operations (Lemma D.7).

B.3 The protocol

Protocol B.6 (Whir/Ligerito)**.**

Parameters. The scheme commits to $f : \{0,1\}^M \to K$ across $\Lambda \geq 1$ *levels* $i = 0, \ldots, \Lambda - 1$ plus a plaintext final level Λ: variable counts $M = M_0 > M_1 > \cdots > M_\Lambda \geq 0$ with fold factors $\ell_i := M_i - M_{i+1} \geq 1$ and M_Λ small (e.g. $M_\Lambda \leq 5$); per level, a rate $\rho_i = 2^{-\nu_i}$, $\nu_i \geq 1$, giving the encoder $\mathrm{Enc}_i := \mathrm{Enc}_{\kappa_i,\nu_i}$ of §B.2 with $\kappa_i := M_{i+1}$, i.e. the code $\mathcal{C}_i := \mathrm{RS}[E, \mathcal{D}_i, 2^{\kappa_i}]$ over its domain $\mathcal{D}_i \subseteq K$ of size $n_i = 2^{\kappa_i+\nu_i}$ (requires $\kappa_i + \nu_i \leq 64$); and a query count ω_i. (The proximity radii $\gamma_i \in (0, 1 - \sqrt{\rho_i})$ appear only in the analysis.)

Commit$(f : \{0,1\}^M \to K)$: **P** sends the oracle $Z^{(0)}[u, \cdot] := \mathrm{Enc}_0(f(u, \cdot))$, $u \in \{0,1\}^{\ell_0}$, whose entries are K elements. No OOD sample is taken.

Open$(\{(W_t, c_t)\}_{t=1,\ldots,J_0})$, $J_0 \geq 1$ weighted claims about f: set $f^{(0)} := f$. For $i = 0, 1, \ldots, \Lambda - 1$:

1. *(batch)* **V** sends $\lambda \leftarrow E$; the *batched claim* is $(W, c^{(0)}) := (\sum_{t=1}^{J_i} \lambda^{t-1} W_t,\ \sum_t \lambda^{t-1} c_t)$.
2. *(sumcheck [LFKN92])* For $j = 1, \ldots, \ell_i$: **P** sends $h_j \in E[X]$ with $\deg h_j \leq 2$, honestly $h_j(X) = \sum_x \widetilde{W}(\chi^{<j}, X, x)\, \widetilde{f^{(i)}}(\chi^{<j}, X, x)$ where $\chi^{<j} := (\chi_1, \ldots, \chi_{j-1})$; **V** checks $h_j(0) + h_j(1) \stackrel{?}{=} c^{(j-1)}$, samples $\chi_j \leftarrow E$ and sends it to the prover; both set $c^{(j)} := h_j(\chi_j)$.
   Set $\chi := (\chi_1, \ldots, \chi_{\ell_i})$, $c' := c^{(\ell_i)}$, and, for $x \in \{0,1\}^{M_{i+1}}$: $f^{(i+1)}(x) := \widetilde{f^{(i)}}(\chi, x)$, $W'(x) := \widetilde{W}(\chi, x)$. The *residual claim* (W', c') is a weighted claim about the folded multilinear $f^{(i+1)} : \{0,1\}^{M_{i+1}} \to E$.
3. If $i < \Lambda - 1$:
   (a) *(commit)* **P** sends the oracle $Z^{(i+1)}[u, \cdot] := \mathrm{Enc}_{i+1}(f^{(i+1)}(u, \cdot))$, $u \in \{0,1\}^{\ell_{i+1}}$.
   (b) *(ood)* **V** sends $z \leftarrow E^{M_{i+1}}$; **P** sends $c_{\mathrm{ood}} \in E$ (honestly $\widetilde{f^{(i+1)}}(z)$).
   (c) *(query)* **V** sends ω_i i.i.d. columns $\mathcal{Q}_i = (x_q)_{q=1,\ldots,\omega_i} \leftarrow \mathcal{D}_i$; it reads $Z^{(i)}[\cdot, x_q]$ and computes $c_q := \sum_u \mathrm{eq}(\chi, u)\, Z^{(i)}[u, x_q] = Z^{(i)}_\chi[x_q]$; honestly $c_q = \mathrm{Enc}_i(f^{(i+1)})[x_q]$, by Fact B.2.
   (d) Claims for level $i + 1$ (so $J_{i+1} := \omega_i + 2$): the residual claim (W', c'), the OOD claim $(\mathrm{eq}(z, \cdot), c_{\mathrm{ood}})$, and the ω_i *consistency claims* $(W^{(i)}_{x_q}, c_q)$, where $W^{(i)}_x$ are the column weights of Enc_i (§B.2): the q-th asserts $\mathrm{Enc}_i(f^{(i+1)})[x_q] = c_q$.
4. If $i = \Lambda - 1$: *(final)* **P** sends the multilinear $f^* : \{0,1\}^{M_\Lambda} \to E$; **V** sends $\omega_{\Lambda-1}$ i.i.d. columns $\mathcal{Q}_{\Lambda-1} \leftarrow \mathcal{D}_{\Lambda-1}$ and computes c_q from $Z^{(\Lambda-1)}$ as in 3(c). Consistency is again a weighted claim, $\mathrm{Enc}_{\Lambda-1}(f^*)[x_q] = \langle W^{(\Lambda-1)}_{x_q},\ f^* \rangle$ (Lemma D.7), so the level closes like the others: **V** samples and sends $\lambda \leftarrow E$, batching the $\omega_{\Lambda-1}$ consistency claims with the residual claim (W', c') into one claim about f^*; M_Λ further sumcheck rounds discharge it, and **V** closes by evaluating $\widetilde{f^*}$ at their point from the table it holds.

V accepts iff every check above passed.

All weights in the protocol are MLE-friendly and multilinear, so every sumcheck summand $\widetilde{W} \cdot \widetilde{f^{(i)}}$ has individual degree at most 2. At level 0 that summand pairs an E-valued weight with the K-valued f; every later level is E throughout.

Completeness is perfect: the honest prover passes every check ($h_j(0) + h_j(1) = c^{(j-1)}$ by definition of h_j, $c_q = \mathrm{Enc}_i(f^{(i+1)})[x_q]$ by Fact B.2, the OOD and residual claims by construction, the final checks as identities).

B.4 Round-by-round soundness

Fix a commitment $\mathsf{cm} = Z^{(0)}$ and input claims $\{(W_t, c_t)\}_t$, and define the relation

$$\mathsf{R}_{\text{open}} := \Big\{ (Z^{(0)}, \{(W_t, c_t)\}_t) \;:\; \exists U \in \mathcal{L}_{\gamma_0}(Z^{(0)}) \text{ with } \langle W_t,\, g_U \rangle = c_t \text{ for all } t \Big\}.$$

The commitment binds the prover to the list $\mathcal{G}_{\mathsf{cm}} := \{\widetilde{g_U} : U \in \mathcal{L}_{\gamma_0}(Z^{(0)})\}$, of size at most $L_0 := \frac{1}{2\eta_0\sqrt{\rho_0}}$ for $\gamma_0 = 1 - \sqrt{\rho_0} - \eta_0$, by the Johnson bound (Theorem B.3).

Theorem B.7 (RBR soundness)**.** *Suppose that for every level i: $\gamma_i = 1 - \sqrt{\rho_i} - \eta_i$ with $\eta_i \in (0, 1 - \sqrt{\rho_i})$, $L_i := \frac{1}{2\eta_i\sqrt{\rho_i}}$, and $\epsilon_i := A_{\text{mca},i}/|E|$ is the mutual correlated agreement error of Theorem B.9 below for $\mathcal{C}_i$ at radius γ_i. Then the Protocol B.6 is a L_0-list binding PCS, and its opening phase R_{open} has the following RBR soundness error at each verifier message:*

$$\begin{aligned}
&\text{batching, level } i: && \varepsilon_i^{\text{batch}} &&= \frac{(J_i - 1)\, L_i}{|E|},\\
&\text{fold challenge } \chi_j,\ \text{level } i: && \varepsilon_{i,j}^{\text{fold}} &&= \frac{2L_i}{|E|} + 2^{\ell_i - j}\,\epsilon_i,\\
&\text{OOD challenge entering level } i \geq 1: && \varepsilon_i^{\text{ood}} &&= \binom{L_i}{2} \cdot \frac{M_i}{|E|},\\
&\text{query message } \mathcal{Q}_i: && \varepsilon_i^{\text{query}} &&= (1 - \gamma_i)^{\omega_i},\\
&\text{final batching challenge } \lambda: && \varepsilon^{\text{fin}} &&= \frac{\omega_{\Lambda-1}}{|E|},\\
&\text{each of the } M_\Lambda \text{ closing rounds}: && \varepsilon^{\text{tail}} &&= \frac{2}{|E|}.
\end{aligned}$$

Here J_0 is the number of input claims and $J_i = \omega_{i-1} + 2$ for $i \geq 1$. The RBR error*, which governs the Fiat–Shamir compilation (§B.1.2), is the maximum of these entries over all rounds.*

The remainder of this section proves the theorem: §B.4.1 states the coding-theoretic tools, §B.4.2 the invariant and the round-by-round argument.

B.4.1 Proximity tools

The engine is *mutual correlated agreement* (MCA) [ACFY25]. For a two-row word Z, recall $Z_\chi = (1 - \chi)Z[0, \cdot] + \chi\, Z[1, \cdot]$.[1]

Theorem B.8 (MCA in the unique-decoding regime; adapted from [BSCH+25, Cor. 1.4])**.** *Let* $\mathrm{RS}[E, \mathcal{D}, k]$ *have block length $n = |\mathcal{D}|$ and relative minimum distance $\delta_{\min} := 1 - (k-1)/n$. Suppose*

$$\delta_{\min} \geq 3\sqrt{\frac{2}{n}} \qquad \text{and} \qquad \gamma \in \left[\frac{\delta_{\min}}{3},\, \frac{\delta_{\min}}{2} - \frac{3}{\delta_{\min} n}\right].$$

Then, for every two-row word Z,

$$\Pr_{\chi \leftarrow E}\left[\exists \mathcal{A} \subseteq \mathcal{D},\ |\mathcal{A}| \geq (1-\gamma)n : \begin{array}{l} Z_\chi \text{ agrees with the code on } \mathcal{A}, \text{ and} \\ Z \text{ does not agree with the interleaved code on } \mathcal{A} \end{array} \right] \leq \frac{\gamma n + 1}{|E|}.$$

[1] [BSCH+25] state the results for the affine line $Z[0, \cdot] + \chi\, Z[1, \cdot]$; in characteristic 2, $Z_\chi = Z[0, \cdot] + \chi(Z[0, \cdot] + Z[1, \cdot])$, and this invertible change of pair maps codeword pairs to codeword pairs, so the formulations are equivalent.

We record Theorem B.8 only for comparison, we don't need it in the List-Decoding Regime, we instead need:

Theorem B.9 (MCA up to the Johnson bound; [BSCH+25, Thm. 4.6])**.** *Let* $\mathrm{RS}[E, \mathcal{D}, k]$ *have block length* $n = |\mathcal{D}|$ *and set* $\rho := (k-1)/n$*, the* slightly reduced rate *([BSCH+25] parametrize by the degree bound* $k-1$*, not the dimension* k*). Let* $\gamma \in (0, 1-\sqrt{\rho})$*, and set* $\eta := 1 - \sqrt{\rho} - \gamma$ *and* $\theta := \max\{\lceil \frac{\sqrt{\rho}}{\eta} \rceil, 3\}$ *(denoted* m *in [BSCH+25]). For every two-row word* Z*,*

$$\Pr_{\chi \leftarrow E}\left[\exists \mathcal{A} \subseteq \mathcal{D},\ |\mathcal{A}| \geq (1-\gamma)n : \begin{array}{l} Z_\chi \text{ agrees with the code on } \mathcal{A}, \text{ and} \\ Z \text{ does not agree with the interleaved code on } \mathcal{A} \end{array} \right] \leq \frac{A_{\mathrm{mca}}}{|E|},$$

where

$$A_{\mathrm{mca}} \;=\; \frac{2(\theta+\frac{1}{2})^5 + 3(\theta+\frac{1}{2})\gamma\rho}{3\rho^{3/2}} \cdot n \;+\; \frac{\theta+\frac{1}{2}}{\sqrt{\rho}}$$

We write $\epsilon := A_{\mathrm{mca}}/|E|$ for this MCA error at radius γ. MCA is exactly what Lemma B.10 needs: folding should not create *new* nearby codewords.

Lemma B.10 (Folding preserves lists)**.** *Let* $\mathrm{RS}[E, \mathcal{D}, k]$ *have rate* ρ*, let* $\gamma \in (0, 1-\rho)$*, and let* ϵ *be an MCA error at radius* γ*. For every interleaved word* $Z \in E^{\{0,1\}^\ell \times \mathcal{D}}$*,* $\ell \geq 1$*:*

$$\Pr_{\chi \leftarrow E}\Big[\, \mathcal{L}_\gamma(Z_\chi) \neq \{U_\chi : U \in \mathcal{L}_\gamma(Z)\} \,\Big] \;\leq\; 2^{\ell-1}\epsilon.$$

Proof. $\supseteq$ holds always: $U \in \mathcal{L}_\gamma(Z)$ agreeing with Z on $\mathcal{A}$ gives U_χ agreeing with Z_χ on $\mathcal{A}$. For $\subseteq$: for $u \in \{0,1\}^{\ell-1}$ let E_u be the MCA event for the two-row word $(Z[(0,u),\cdot], Z[(1,u),\cdot])$, so $\Pr[\bigcup_u E_u] \leq 2^{\ell-1}\epsilon$. Suppose no E_u occurs, and let $V \in \mathcal{L}_\gamma(Z_\chi)$ with common agreement set $\mathcal{A}$, $|\mathcal{A}| \geq (1-\gamma)n$. For each u, the row $Z_\chi[u,\cdot]$ agrees on $\mathcal{A}$ with the codeword $V[u,\cdot]$, so (as E_u failed) there are, for each $\iota \in \{0,1\}$, codewords $W_{(\iota,u)}$ equal to $Z[(\iota,u),\cdot]$ on $\mathcal{A}$; the interleaved codeword W with these rows lies in $\mathcal{L}_\gamma(Z)$ (same $\mathcal{A}$ for all u). Finally $W_\chi[u,\cdot]$ agrees with $V[u,\cdot]$ on $\mathcal{A}$, and $|\mathcal{A}| \geq (1-\gamma)n > k$ while distinct codewords agree on fewer than k points, so $V = W_\chi$. □

Lemma B.11 (OOD separation)**.** *Let* Z *be an interleaved word with* $|\mathcal{L}_\gamma(Z)| \leq L$*, decoding to* M*-variable multilinears. Then* $\Pr_{z \leftarrow E^M}[\exists$ *distinct* $U, U' \in \mathcal{L}_\gamma(Z) : \widetilde{g_U}(z) = \widetilde{g_{U'}}(z)] \leq \binom{L}{2} M/|E|$.

Proof. Distinct interleaved codewords have distinct multilinear extensions; apply Lemma 3.8 to each difference and union bound. □

B.4.2 Proof of the main theorem

At level i, abbreviate $\chi^{\leq j} := (\chi_1, \ldots, \chi_j)$ and $\chi^{<j} := \chi^{\leq j-1}$; thus $\chi^{\leq 0}$ is empty and $Z^{(i)}_{\chi^{\leq 0}} = Z^{(i)}$. For $0 \leq j \leq \ell_i$, the *round-j claim* is the weighted claim

$$(\, \widetilde{W}(\chi^{\leq j}, \cdot),\ c^{(j)} \,) \qquad \text{about multilinears } g : \{0,1\}^{M_i - j} \to E,$$

where W, $c^{(0)}$, and $c^{(j)} = h_j(\chi_j)$ are read off the transcript as in Protocol B.6; an interleaved codeword satisfies a claim if its decoded multilinear does. The round-0 claim is the batched claim, and the round-ℓ_i claim is the residual claim (W', c').

We now define the invariant required by Definition B.5, a condition on the partial transcripts of the opening. In one phrase: *every codeword close to the current fold violates the current claim.* Precisely, the list below gives the condition just after each verifier message; a transcript ending with prover messages inherits the condition of its last verifier message. Every condition also includes one implicit disjunct, the *escape clause*: *"some verifier check has already failed"* (such transcripts are rejected no matter what follows).

- *Start of level i* (before any message for $i = 0$, after the query message J_{i-1} otherwise): every $U \in \mathcal{L}_{\gamma_i}(Z^{(i)})$ violates at least one of the level's J_i claims (the input claims for $i = 0$, the claims of step 3(d) of Protocol B.6 otherwise).
- *After the batching challenge λ of level i, and after each of its fold challenges χ_j:* every $U \in \mathcal{L}_{\gamma_i}(Z^{(i)}_{\chi \le j})$ violates the round-j claim (λ being the case $j = 0$).
- *After the OOD challenge z closing level i ($i < \Lambda - 1$):* the round-ℓ_i instance above holds, and the multilinear extensions decoded from $\mathcal{L}_{\gamma_{i+1}}(Z^{(i+1)})$ are pairwise distinct at z.
- *After the final query message $\mathcal{Q}_{\Lambda-1}$:* f^* violates one of the level's $\omega_{\Lambda-1} + 1$ claims.
- *After the final λ, and after each of the M_Λ rounds that follow (complete transcript):* f^* violates the running claim. The terminal one $\mathbf{V}$ checks against $\widetilde{f^*}$ itself, so it rejects.

Conditions (i), (ii), (iv) of Definition B.5 hold by inspection: on a false statement, the first item is exactly the failure of R_{open}; a prover message changes nothing an instance depends on, except possibly failing a check, which only switches the escape clause on; and the last item is precisely "some check failed", so such a complete transcript is rejected. The separation clause of the OOD instance is there because the query message consumes it: it pins the new oracle's list to at most one candidate consistent with the prover's answer. It remains to bound the escape probabilities (iii); in the proof below we assume all checks so far pass, since otherwise the escape clause already holds.

Proof of Theorem B.7. If the statement lies outside R_{open}, then before any message is sent every $U \in \mathcal{L}_{\gamma_0}(Z^{(0)})$ violates at least one input claim, so the invariant holds. We follow the protocol and bound, for each verifier message, the probability that the invariant fails to carry over. Every list below consists of interleaved words of $\mathcal{C}_i$ within radius γ_i, hence has size at most L_i by Theorem B.3.

Batching challenge λ. Assume that every $U \in \mathcal{L}_{\gamma_i}(Z^{(i)})$ violates at least one of the J_i claims (W_t, c_t). Fix such a U. Its discrepancy against the batched claim is $\sum_t \lambda^{t-1} d_t$ with $d_t := \langle W_t, g_U \rangle - c_t$, a polynomial in λ of degree less than J_i, nonzero because the d_t are not all zero. By Lemma 3.8 it vanishes with probability at most $(J_i - 1)/|E|$, so a union bound over the at most L_i list members leaves every U violating the round-0 claim except with probability $(J_i - 1)L_i/|E|$.

Fold challenge χ_j. Assume that every $U \in \mathcal{L} := \mathcal{L}_{\gamma_i}(Z^{(i)}_{\chi^{<j}})$ violates the round-$(j-1)$ claim. Since all checks so far pass (otherwise the verifier trivially rejects), $h_j(0) + h_j(1) = c^{(j-1)}$. For $U \in \mathcal{L}$, let

$$H_U(X) \;:=\; \sum_{x \in \{0,1\}^{M_i - j}} \widetilde{W}(\chi^{<j}, X, x)\, \widetilde{g_U}(X, x)$$

be the honest round polynomial for U; it has degree at most 2, since $\widetilde{W}$ and $\widetilde{g_U}$ are multilinear. Summing over $X \in \{0, 1\}$ turns the multilinear evaluations back into values, so

$$H_U(0) + H_U(1) \;=\; \langle\, \widetilde{W}(\chi^{<j}, \cdot),\; g_U \,\rangle \;\neq\; c^{(j-1)} \;=\; h_j(0) + h_j(1),$$

the inequality being the assumed violation; hence $h_j \neq H_U$ as polynomials. Two bad events over χ_j:

- B_1: $h_j(\chi_j) = H_U(\chi_j)$ for some $U \in \mathcal{L}$. For each U, two distinct polynomials of degree at most 2 agree on at most 2 points, so $\Pr[B_1] \le 2L_i/|E|$.
- B_2: $\mathcal{L}_{\gamma_i}(Z^{(i)}_{\chi \le j}) \neq \{U_{\chi_j} : U \in \mathcal{L}\}$, i.e. folding does not map the old list onto the new one. The matrix $Z^{(i)}_{\chi^{<j}}$ has $2^{\ell_i - j + 1}$ rows, so Lemma B.10 with $\ell = \ell_i - j + 1$ gives $\Pr[B_2] \le 2^{\ell_i - j} \epsilon_i$.

If neither occurs, every $V \in \mathcal{L}_{\gamma_i}(Z^{(i)}_{\chi^{\le j}})$ is $V = U_{\chi_j}$ for some $U \in \mathcal{L}$, with $\widetilde{g_V} = \widetilde{g_U}(\chi_j, \cdot)$ by Fact B.2, so

$$\langle\, \widetilde{W}(\chi^{\le j}, \cdot),\ g_V \,\rangle \;=\; \sum_{x \in \{0,1\}^{M_i - j}} \widetilde{W}(\chi^{<j}, \chi_j, x)\, \widetilde{g_U}(\chi_j, x) \;=\; H_U(\chi_j) \;\neq\; h_j(\chi_j) \;=\; c^{(j)},$$

using $\neg B_1$ for the inequality: every V violates the round-j claim. The total error is $2L_i/|E| + 2^{\ell_i - j}\epsilon_i$.

The level interface. After the last fold challenge of level $i < \Lambda - 1$, we know: every codeword within γ_i of the single row $c := Z^{(i)}_{\chi} \in E^{\mathcal{D}_i}$ violates the residual claim (W', c'). The prover now sends the new oracle $Z^{(i+1)}$.

OOD challenge z. After z, two properties must hold: the one above, which does not involve z; and pairwise distinctness at z of the multilinear extensions decoded from $\mathcal{L}_{\gamma_{i+1}}(Z^{(i+1)})$, a list already determined by the prover's message. The latter fails with probability at most $\binom{L_{i+1}}{2} M_{i+1}/|E|$ over z, by Lemma B.11.

Query message $\mathcal{Q}_i$. Assume both properties of the previous step; the prover has since sent its answer c_{ood}. The values $\widetilde{g_U}(z)$ are pairwise distinct across $\mathcal{L}_{\gamma_{i+1}}(Z^{(i+1)})$, so at most one member U^* can satisfy $\widetilde{g_{U^*}}(z) = c_{\text{ood}}$: every other member violates the OOD claim $(\text{eq}(z, \cdot), c_{\text{ood}})$, whatever $\mathcal{Q}_i$ is. If no such U^* exists, we are done. Otherwise write $g^* := g_{U^*}$, a multilinear on $\{0,1\}^{M_{i+1}}$, and distinguish two cases.

- $\text{Enc}_i(g^*)$ *agrees with c on at least $(1 - \gamma_i)n_i$ coordinates.* Then the codeword $\text{Enc}_i(g^*)$ lies in $\mathcal{L}_{\gamma_i}(c)$, hence violates the residual claim by assumption: $\langle W',\, g^* \rangle \neq c'$, whatever $\mathcal{Q}_i$ is.
- *Otherwise.* A uniform column $x \leftarrow \mathcal{D}_i$ satisfies $\text{Enc}_i(g^*)[x] = c[x]$ with probability less than $1 - \gamma_i$, so all ω_i i.i.d. queries do so simultaneously with probability less than $(1 - \gamma_i)^{\omega_i}$; and any disagreeing query x_q makes U^* violate the consistency claim $(W^{(i)}_{x_q},\, c_q)$, since $c_q = c[x_q]$.

Except with probability $(1 - \gamma_i)^{\omega_i}$, every member of $\mathcal{L}_{\gamma_{i+1}}(Z^{(i+1)})$ thus violates at least one of the J_{i+1} claims, which is exactly the assumption of the batching step at level $i + 1$.

Final level. At level $\Lambda - 1$ the fold challenges end as above: every codeword within $\gamma_{\Lambda-1}$ of $c := Z^{(\Lambda-1)}_{\chi}$ violates the residual claim. The prover sends f^*. If $\text{Enc}_{\Lambda-1}(f^*)$ agrees with c on at least $(1 - \gamma_{\Lambda-1})n_{\Lambda-1}$ coordinates then $\text{Enc}_{\Lambda-1}(f^*) \in \mathcal{L}_{\gamma_{\Lambda-1}}(c)$, so f^* violates the residual claim; otherwise, except with probability $(1 - \gamma_{\Lambda-1})^{\omega_{\Lambda-1}}$, some query disagrees and f^* violates that consistency claim. Either way it violates one of the $\omega_{\Lambda-1} + 1$ batched claims, hence the batch itself except with probability $\omega_{\Lambda-1}/|E|$ (Lemma 3.8), with no union over a list: f^* is transmitted, not one of the codewords near an oracle. The M_Λ rounds that follow are a plain sumcheck (Fact 3.10) on a polynomial that $\mathbf{V}$ holds, so the terminal check against $\widetilde{f^*}$ fails.

□

B.5 Optimizations

The committed f is the stack of §4.1: N_{stack} field elements in K, then zeros up to 2^M. Let the row index be the stack's most significant ℓ_0 coordinates: $f(u, x) := q(x, u)$ for $u \in \{0,1\}^{\ell_0}$ and $x \in \{0,1\}^{M - \ell_0}$, §D. Row u is then the block of $2^{M-\ell_0}$ consecutive elements of q at offset $\langle u \rangle 2^{M-\ell_0}$, the padding is whole rows, and only $N_{\text{rows}} = \lceil N_{\text{stack}}/2^{M-\ell_0} \rceil$ of the 2^{ℓ_0} rows hold data. So we get the following optimizations at commitment:

- the FFT-encoder only runs on N_{rows} rows (and not the full 2^{ℓ_0}), since FFT-ing zeros gives zeros;

- a Merkle leaf only contains N_{rows} elements of K (proof size optimization), we keep the convention of hashing the full 2^{ℓ_0} values for simplicity, but we hash them in reverse order, starting from the padding zeros;
- the prover can thus preprocess the internal state of BLAKE2s after hashing those zeros, and then hash the useful part of each leaf starting from this state.

Conclusion: at commitment, both the FFT and the Merkle leaf hashing cost are proportional to N_{stack} (not its next power of two). Unfortunately we still pay for the padding inside the N_{stack} field elements, because some instruction counts are not powers of two. This could be prevented using the Jagged PCS [HJR+25].

B.6 The Johnson bound

We prove Theorem B.3 in its general form (arbitrary code).

Theorem B.12 (Johnson bound, agreement form)**.** *Let $\mathcal{C} \subseteq \Sigma^n$ be a code, over any alphabet Σ, in which two distinct codewords agree on at most ρn coordinates, and let $\eta \in (0, 1-\sqrt{\rho})$. Then every word $\mathbf{c} \in \Sigma^n$ has at most $\frac{1}{2\eta\sqrt{\rho}}$ codewords agreeing with it on at least $(\sqrt{\rho}+\eta)\,n$ coordinates.*

Proof. Let $U_1, \dots, U_L$ be distinct codewords, each agreeing with $\mathbf{c}$ on at least $(\sqrt{\rho}+\eta)n$ coordinates. For each j, fix a set $\mathcal{A}_j$ of exactly $a := \lceil(\sqrt{\rho}+\eta)n\rceil$ coordinates on which U_j agrees with $\mathbf{c}$, and write $\varphi := a/n$, so $\sqrt{\rho}+\eta \le \varphi \le 1$. For a coordinate i, let $d_i := |\{j : i \in \mathcal{A}_j\}|$; then $\sum_i d_i = La = L\varphi n$.

Wherever two codewords both agree with $\mathbf{c}$ they agree with each other, so $|\mathcal{A}_j \cap \mathcal{A}_{j'}| \le \rho n$ for $j \ne j'$, and counting ordered pairs,

$$\sum_i d_i(d_i-1) \;=\; \sum_{j\ne j'} |\mathcal{A}_j \cap \mathcal{A}_{j'}| \;\le\; L(L-1)\,\rho n.$$

Cauchy–Schwarz on the vectors $(d_1, \dots, d_n)$ and $(1, \dots, 1)$ gives $\left(\sum_i d_i\right)^2 \le n \sum_i d_i^2$, i.e. $\sum_i d_i^2 \ge (L\varphi n)^2/n = L^2\varphi^2 n$; the left side above, which equals $\sum_i d_i^2 - \sum_i d_i$, is therefore at least $L^2\varphi^2 n - L\varphi n$. Dividing by Ln and rearranging,

$$L\varphi^2 - \varphi \;\le\; (L-1)\,\rho \qquad\Longleftrightarrow\qquad L\,(\varphi^2-\rho) \;\le\; \varphi - \rho.$$

Finally $\varphi^2 - \rho \ge (\sqrt{\rho}+\eta)^2 - \rho = 2\eta\sqrt{\rho} + \eta^2 > 2\eta\sqrt{\rho}$ while $\varphi - \rho \le 1$, so $L \le \frac{1}{2\eta\sqrt{\rho}}$. □

C The Flock protocol

Flock [BRW26] proves many executions of one Boolean circuit at once, in our case the BLAKE2s compression. Since we do not follow their protocol exactly, we describe our version here.

C.1 Statement and commitment

One BLAKE2s compression has a Boolean R1CS witness block of $2^{\kappa_{\text{block}}}$ bits, where $\kappa_{\text{block}} = 14$. Its fixed matrices are $A_0, B_0, C_0 \in \mathbb{F}_2^{2^{\kappa_{\text{block}}} \times 2^{\kappa_{\text{block}}}}$, with $C_0 = I$. The witness includes the circuit inputs, outputs, AND-gate values, addition carries and one constant-one position; linear wires are substituted away. Within-block coordinates are the low-order coordinates and batch coordinates are the high-order coordinates throughout this annex, and a Boolean tuple is also read as an integer, lowest coordinate first.

For $2^{\kappa_{\text{batch}}}$ independent compressions, the witness is:

$$z : \{0,1\}^{\kappa_{\text{bool}}} \longrightarrow \mathbb{F}_2, \qquad \kappa_{\text{bool}} = \kappa_{\text{block}} + \kappa_{\text{batch}},$$

and the batched matrices are

$$A = I_{2^{\kappa_{\text{batch}}}} \otimes A_0, \qquad B = I_{2^{\kappa_{\text{batch}}}} \otimes B_0, \qquad C = I_{2^{\kappa_{\text{bool}}}}$$

Writing $a = Az$, $b = Bz$ and $c = Cz = z$, the R1CS statement is:

$$a(u)b(u) + c(u) = 0 \qquad \text{for every } u \in \{0,1\}^{\kappa_{\text{bool}}} \tag{10}$$

We also enforce the position 512 of every block to be 1 (preventing the all-zero witness):

$$z(0,0,0,0,0,0,0,0,0,1,0,0,0,0,\, t) = 1 \qquad \text{for every } t \in \{0,1\}^{\kappa_{\text{batch}}} \tag{11}$$

This second condition is enforced by lincheck (§C.3), and position $512 = j_{\mathbf{1}}$ is where every circuit below puts its constant wire. Since it is asked of every block, the prover pads the batch out to $2^{\kappa_{\text{batch}}}$ with copies of a genuine witness block rather than with zeros.

Set $\kappa_{\text{skip}} = 6$ and $\kappa_{\text{flock}} = \kappa_{\text{bool}} - \kappa_{\text{skip}}$. The prover commits to z by packing its low κ_{skip} coordinates, 64 bits, to a K-element, following Ring-Switching (Annex A). The resulting multilinear $q_{\text{flock}} : \{0,1\}^{\kappa_{\text{flock}}} \to K$ is committed as one region of the stack (§4.1).

C.2 Zerocheck

Flock zerochecks (10) (Definition 3.11), but replaces the first κ_{skip} rounds by one univariate round [Gru24]. Split $u = (i, v)$ with $i \in \{0,1\}^{\kappa_{\text{skip}}}$ and $v \in \{0,1\}^{\kappa_{\text{flock}}}$. Let $\varphi_8 : \mathbb{F}_{2^8} \hookrightarrow E$ be the subfield embedding, with a byte interpreted in the polynomial basis, and define

$$\mathcal{H} = \{\varphi_8(0), \ldots, \varphi_8(63)\}, \qquad \mathcal{H}' = \varphi_8(64) + \mathcal{H} = \{\varphi_8(64), \ldots, \varphi_8(127)\}.$$

Thus $\mathcal{H}$ is a 64-point additive subspace and $\mathcal{H}'$ is its disjoint coset, in the additive-NTT ordering of Annex B.

Write ϖ_i for the Lagrange basis of $\mathcal{H}$: $\varpi_i(\varphi_8(i')) = [i = i']$.

Definition C.1 (Quirky extension)**.** For $f : \{0,1\}^{\kappa_{\text{skip}}} \times \{0,1\}^n \to \mathbb{F}_2$, its *quirky extension* interpolates the first κ_{skip} coordinates over $\mathcal{H}$ and extends the other n multilinearly:

$$f^\sharp : E \times E^n \to E, \qquad f^\sharp(Y, v) := \sum_{i=0}^{63} \varpi_i(Y)\, \widetilde{f}(i, v),$$

so $f^\sharp$ has degree 63 in Y, degree one in every other variable, and $f^\sharp(\varphi_8(i), v) = \widetilde{f}(i, v)$ for every $i \in \{0,1\}^{\kappa_{\text{skip}}}$. Below f is a, b or c, with $n = \kappa_{\text{flock}}$, or A_0 or B_0 read as a row followed by a column, with $n = 2\kappa_{\text{block}} - \kappa_{\text{skip}}$. Its argument (Y, v) is a *quirky point*.

The zerocheck point $r \in E^{\kappa_{\text{flock}}}$ is defined as follows:

- Its first seven coordinates $r_0, \ldots, r_6$ are fixed to $\varphi_8(\texttt{0xf7}), \varphi_8(\texttt{0x53}), \varphi_8(\texttt{0xb5})$ and $g_0^{2^j}/(1+g_0^{2^j})$ for $0 \le j < 4$, where $g_0 \in E$ is public. Lemma 3.12 applies.
- The remaining coordinates $r_7, \ldots, r_{\kappa_{\text{flock}}-1}$ are sampled at random by the verifier.

Define

$$P(Y) = \sum_{v \in \{0,1\}^{\kappa_{\text{flock}}}} \text{eq}(r, v)\big(a^\sharp(Y, v)b^\sharp(Y, v) + c^\sharp(Y, v)\big), \tag{12}$$

which lies in $E[Y]$ and has degree at most 126, the product contributing 2×63 and the linear term 63. If (10) holds, then $P(h) = 0$ for every $h \in \mathcal{H}$. The prover sends $P|_{\mathcal{H}'}$, 64 values: together with the 64 assumed zeros on $\mathcal{H}$ that is a degree-below-128 polynomial's worth of evaluations, so the verifier interpolates P over the 128-point space $\mathcal{H} \cup \mathcal{H}'$.

The verifier then samples $\upsilon_{\text{zc}} \in E$ and reconstructs the claimed value

$$v_P \stackrel{?}{=} P(\upsilon_{\text{zc}}).$$

The parties run sumcheck over the remaining κ_{flock} Boolean variables in (12), with Gruen's eq-factor optimization [Gru24]. Writing $R_{\text{zc}} \in E$ for the running claim after the last round, the prover sends $v_a, v_b \in E$ and the terminal identity

$$R_{\text{zc}} = v_a v_b + v_c \tag{13}$$

is what *defines* v_c: both sides solve it rather than transmit it. The zerocheck therefore ends leaving three claims at one point,

$$a^\sharp(\upsilon_{\text{zc}}, \chi) = v_a, \qquad b^\sharp(\upsilon_{\text{zc}}, \chi) = v_b, \qquad z^\sharp(\upsilon_{\text{zc}}, \chi) = v_c, \tag{14}$$

where $\chi \in E^{\kappa_{\text{flock}}}$ is the vector of sumcheck challenges and the third claim uses $c = z$.

C.3 Lincheck

Lincheck reduces the three zerocheck claims to claims on z. The block structure is what keeps it cheap: the matrices repeat one BLAKE2s circuit, so lincheck runs over a single block, not over the batch. Split $\chi = (\chi_{\text{in}}, \chi_{\text{out}})$, where $\chi_{\text{in}} \in E^{\kappa_{\text{block}} - \kappa_{\text{skip}}}$ selects the remaining within-block coordinates and $\chi_{\text{out}} \in E^{\kappa_{\text{batch}}}$ selects a batch instance.

Lemma C.2 (Block diagonality folds the batch into z)**.** *For $a = Az$ with $A = I_{2^{\kappa_{\text{batch}}}} \otimes A_0$, at every point $(\upsilon_{\text{zc}}, \chi_{\text{in}}, \chi_{\text{out}}) \in E \times E^{\kappa_{\text{block}} - \kappa_{\text{skip}}} \times E^{\kappa_{\text{batch}}}$,*

$$a^\sharp(\upsilon_{\text{zc}}, \chi_{\text{in}}, \chi_{\text{out}}) = \sum_{j \in \{0,1\}^{\kappa_{\text{block}}}} A_0^\sharp(\upsilon_{\text{zc}}, \chi_{\text{in}}, j)\, \widetilde{z}(j, \chi_{\text{out}}),$$

and likewise for $b = Bz$ with B_0 in place of A_0.

Proof. Block diagonality gives

$$a(k, t) = \sum_{j \in \{0,1\}^{\kappa_{\text{block}}}} A_0(k, j)\, z(j, t) \qquad \text{for } k \in \{0,1\}^{\kappa_{\text{block}}},\ t \in \{0,1\}^{\kappa_{\text{batch}}},$$

so, splitting the row as $k = (k_{\text{sk}}, k_{\text{in}})$ with $k_{\text{sk}} \in \{0,1\}^{\kappa_{\text{skip}}}$ and $k_{\text{in}} \in \{0,1\}^{\kappa_{\text{block}} - \kappa_{\text{skip}}}$ and expanding

every extension by Fact 3.6,

$$\begin{aligned}
a^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, \chi_{\mathrm{out}}) &= \sum_{k_{\mathrm{sk}} \in \{0,1\}^{\kappa_{\mathrm{skip}}}} \varpi_{k_{\mathrm{sk}}}(\upsilon_{\mathrm{zc}})\, \widetilde{a}(k_{\mathrm{sk}}, \chi_{\mathrm{in}}, \chi_{\mathrm{out}}) \\
&= \sum_{k_{\mathrm{sk}}} \varpi_{k_{\mathrm{sk}}}(\upsilon_{\mathrm{zc}}) \sum_{k_{\mathrm{in}}} \mathrm{eq}(\chi_{\mathrm{in}}, k_{\mathrm{in}}) \sum_{t \in \{0,1\}^{\kappa_{\mathrm{batch}}}} \mathrm{eq}(\chi_{\mathrm{out}}, t)\, a((k_{\mathrm{sk}}, k_{\mathrm{in}}), t) \\
&= \sum_{k_{\mathrm{sk}}, k_{\mathrm{in}}} \varpi_{k_{\mathrm{sk}}}(\upsilon_{\mathrm{zc}})\, \mathrm{eq}(\chi_{\mathrm{in}}, k_{\mathrm{in}}) \sum_{j \in \{0,1\}^{\kappa_{\mathrm{block}}}} A_0((k_{\mathrm{sk}}, k_{\mathrm{in}}), j) \underbrace{\sum_t \mathrm{eq}(\chi_{\mathrm{out}}, t)\, z(j,t)}_{= \widetilde{z}(j, \chi_{\mathrm{out}})} \\
&= \sum_{j \in \{0,1\}^{\kappa_{\mathrm{block}}}} \underbrace{\Big(\sum_{k_{\mathrm{sk}}, k_{\mathrm{in}}} \varpi_{k_{\mathrm{sk}}}(\upsilon_{\mathrm{zc}})\ \mathrm{eq}(\chi_{\mathrm{in}}, k_{\mathrm{in}})\, A_0((k_{\mathrm{sk}}, k_{\mathrm{in}}), j) \Big)}_{= A_0^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, j)} \widetilde{z}(j, \chi_{\mathrm{out}}).
\end{aligned}$$

□

The c claim has the same shape, with the column index split as $j = (j_{\mathrm{sk}}, j_{\mathrm{in}}) \in \{0,1\}^{\kappa_{\mathrm{skip}}} \times \{0,1\}^{\kappa_{\mathrm{block}} - \kappa_{\mathrm{skip}}}$, j_{sk} the coordinates the univariate skip replaced and j_{in} the within-block coordinates that remain:

$$z^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, \chi_{\mathrm{out}}) = \sum_{j \in \{0,1\}^{\kappa_{\mathrm{block}}}} \varpi_{j_{\mathrm{sk}}}(\upsilon_{\mathrm{zc}})\ \mathrm{eq}(\chi_{\mathrm{in}}, j_{\mathrm{in}})\, \widetilde{z}(j, \chi_{\mathrm{out}}), \tag{15}$$

The verifier samples $\alpha_{\mathrm{lc}} \in E$ and batches the zerocheck's three claims (14) with the constant-position check (11):

$$\begin{aligned}
v_a + \alpha_{\mathrm{lc}} v_b + \alpha_{\mathrm{lc}}^2 v_c + \alpha_{\mathrm{lc}}^3 \stackrel{?}{=} \sum_{j \in \{0,1\}^{\kappa_{\mathrm{block}}}} \Big(& A_0^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, j) + \alpha_{\mathrm{lc}} B_0^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, j) \\
& + \alpha_{\mathrm{lc}}^2 \varpi_{j_{\mathrm{sk}}}(\upsilon_{\mathrm{zc}})\, \mathrm{eq}(\chi_{\mathrm{in}}, j_{\mathrm{in}}) + \alpha_{\mathrm{lc}}^3 [j = 512] \Big) \widetilde{z}(j, \chi_{\mathrm{out}}).
\end{aligned} \tag{16}$$

The parties start by running $\kappa_{\mathrm{block}} - \kappa_{\mathrm{skip}} = 14 - 6 = 8$ sumcheck rounds on (16), binding the j_{in} coordinates (leaving the skip index j_{sk} unfolded), with challenges χ'_{in} in coordinate order (the reverse of their round order). Let $R_{\mathrm{lc}} \in E$ be the resulting sumcheck claim.

The prover sends $(s_0, \ldots, s_{63}) \in E^{64}$, claiming $s_i = \widetilde{z}(i, \chi'_{\mathrm{in}}, \chi_{\mathrm{out}})$. These 64 claims will later be proven by Ring-Switching.

The verifier finally needs to check:

$$\begin{aligned}
R_{\mathrm{lc}} \stackrel{?}{=} \sum_{i=0}^{63} \Big(& A_0^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, i, \chi'_{\mathrm{in}}) + \alpha_{\mathrm{lc}}\, B_0^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, i, \chi'_{\mathrm{in}}) \\
& + \alpha_{\mathrm{lc}}^2 \varpi_i(\upsilon_{\mathrm{zc}})\ \mathrm{eq}(\chi_{\mathrm{in}}, \chi'_{\mathrm{in}}) + \alpha_{\mathrm{lc}}^3 [i = 0]\ \mathrm{eq}(\chi'_{\mathrm{in}}, 8) \Big) s_i.
\end{aligned} \tag{17}$$

C.3.1 Concluding Lincheck for a native verifier

With the row split $k = (k_{\mathrm{sk}}, k_{\mathrm{in}}) \in \{0,1\}^{\kappa_{\mathrm{skip}}} \times \{0,1\}^{\kappa_{\mathrm{block}} - \kappa_{\mathrm{skip}}}$ of the proof above,

$$A_0^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, j) = \sum_{k \in \{0,1\}^{\kappa_{\mathrm{block}}}} e_{\mathrm{row}}(k)\, A_0(k, j), \qquad e_{\mathrm{row}}(k_{\mathrm{sk}}, k_{\mathrm{in}}) = \varpi_{k_{\mathrm{sk}}}(\upsilon_{\mathrm{zc}})\ \mathrm{eq}(\chi_{\mathrm{in}}, k_{\mathrm{in}}), \tag{18}$$

and likewise for B_0.

Writing $w_{\mathrm{col}}(j_{\mathrm{sk}}, j_{\mathrm{in}}) = s_{j_{\mathrm{sk}}} \operatorname{eq}(\chi'_{\mathrm{in}}, j_{\mathrm{in}})$ with $(j_{\mathrm{sk}}, j_{\mathrm{in}}) \in \{0,1\}^{\kappa_{\mathrm{skip}}} \times \{0,1\}^{\kappa_{\mathrm{block}} - \kappa_{\mathrm{skip}}}$, each matrix term in (17) is simplified as follows:

$$\sum_{i=0}^{63} A_0^{\sharp}(v_{\mathrm{zc}}, \chi_{\mathrm{in}}, i, \chi'_{\mathrm{in}})\, s_i = \sum_{k,j \in \{0,1\}^{\kappa_{\mathrm{block}}}} A_0(k,j)\, e_{\mathrm{row}}(k)\, w_{\mathrm{col}}(j). \tag{19}$$

One forward walk of the BLAKE2s circuit returns (19) for both matrices (§C.5), never touching the roughly 89 million nonzero entries they hold. Its committed wires are 896 free inputs, the constant, 256 committed $\oplus$ wires carrying the output chaining value, and 14,720 $\wedge$ wires, one per product bit of the 320 modular additions of the ten rounds; at one multiplication per committed wire on the A side, one per $\wedge$ wire on the B side, and one for the constant factor the remaining B rows share, that is 30,594 field multiplications in $O(\text{circuit})$ work. The c term of (17) is $\alpha_{\mathrm{lc}}^2 \langle e_{\mathrm{row}}, w_{\mathrm{col}} \rangle$ in the same notation, and needs no walk: both sides are tensors, so it contracts to $\alpha_{\mathrm{lc}}^2 \operatorname{eq}(\chi_{\mathrm{in}}, \chi'_{\mathrm{in}}) \sum_i \varpi_i(v_{\mathrm{zc}}) s_i$.

C.3.2 Concluding Lincheck for a recursive verifier

To avoid the costly circuit-walk in recursion, the recursion program receives as hint:

$$M_{\mathrm{lc}} = \sum_{k,j \in \{0,1\}^{\kappa_{\mathrm{block}}}} (A_0(k,j) + \alpha_{\mathrm{lc}} B_0(k,j)) e_{\mathrm{row}}(k)\, w_{\mathrm{col}}(j). \tag{20}$$

This makes it possible to cheaply verify (17). Ensuring validity of the hint is deferred outside of the recursion program, via sumcheck reduction, as described in §10.1.3.

C.3.3 Ring switching

The role of Ring-Switching is now to prove:

$$s_i \overset{?}{=} \widetilde{z}(i, \chi'_{\mathrm{in}}, \chi_{\mathrm{out}}) \qquad \text{for } i \in \{0,1\}^{\kappa_{\mathrm{skip}}}.$$

C.4 Protocol summary

Protocol C.3 (Flock summary)**.** After the witness z has been committed, and its size announced:

1. The verifier forms the zerocheck point r: seven fixed coordinates, the rest sampled. The κ_{skip} variables the univariate skip replaced consume no equality challenge.
2. The prover sends $P|_{\mathcal{H}'}$, 64 values. The verifier samples v_{zc} and reconstructs v_P.
3. The parties run κ_{flock} quadratic zerocheck rounds on (12), which the prover ends by sending v_a, v_b; the terminal identity (13) then fixes v_c.
4. The verifier samples α_{lc} and the parties run the 8 quadratic lincheck rounds on (16). The prover sends $(s_0, \ldots, s_{63})$, and the verifier checks (17): one circuit walk natively, the hint (20) in recursion.

What Flock leaves is one family of 64 claims for Ring-Switching (§C.3.3): $(s_i \overset{?}{=} \widetilde{z}(i, \chi'_{\mathrm{in}}, \chi_{\mathrm{out}}))_{0 \le i \le 63}$.

C.5 Evaluating the matrices

A_0 and B_0 encode a Boolean circuit, but are never materialized.

C.5.1 The circuit

Definition C.4 (Circuit)**.** A *circuit* is an ordered sequence of wires $w_1, \dots, w_N$. Each wire is either a *source*, a $\oplus$ wire with a possibly empty set S_w of earlier wires, or a $\wedge$ wire with an ordered pair (u_w, u'_w) of earlier wires. Its value lies in $\mathbb{F}_2$: a $\oplus$ wire is the sum of the values in S_w, and a $\wedge$ wire is the product of the values of u_w and u'_w. The source w_1 is the constant 1 and the other sources are free inputs.

A set of *committed* wires containing all sources, all $\wedge$ wires, and a subset of the $\oplus$ wires (typically the result of the computation) is equipped with an injection $w \mapsto k(w)$ into $\{0,1\}^{\kappa_{\text{block}}}$. The constant has position $k(w_1) = j_{\mathbf{1}}$. A position $k \in \{0,1\}^{\kappa_{\text{block}}}$ is *empty* if $k \neq k(w)$ for every committed wire w. A witness block $z_t = z(\cdot, t)$ stores each committed wire's value at its position and stores 0 at empty positions.

For $p \in \{0,1\}^{\kappa_{\text{block}}}$, write $[\,\cdot = p\,] : \{0,1\}^{\kappa_{\text{block}}} \to \mathbb{F}_2$ for the function that equals 1 at p and 0 at every other position. The *expansion* of a wire over committed positions is

$$\mathcal{F}_w = [\,\cdot = k(w)\,] \;\text{ if } w \text{ is committed}, \qquad \mathcal{F}_w = \sum_{u \in S_w} \mathcal{F}_u \;\text{ otherwise}, \tag{21}$$

where the second case applies only to uncommitted $\oplus$ wires. By induction, $\langle \mathcal{F}_w,\, z_t \rangle$ is the value of w.

C.5.2 The R1CS matrices

For every committed wire w, assign two sets of wires to the corresponding row:

$$(S_w^A, S_w^B) = \begin{cases} (\{u_w\}, \{u'_w\}) & \text{if } w \text{ is a } \wedge \text{ wire}, \\ (S_w, \{w_1\}) & \text{if } w \text{ is a } \oplus \text{ wire}, \\ (\{w\}, \{w_1\}) & \text{if } w \text{ is a source}. \end{cases}$$

and define:

$$A_0(k(w), \cdot) = \sum_{u \in S_w^A} \mathcal{F}_u, \qquad B_0(k(w), \cdot) = \sum_{u \in S_w^B} \mathcal{F}_u. \tag{22}$$

For every empty position k, set $A_0(k, \cdot) = B_0(k, \cdot) = 0$. This specifies every row of both matrices.

The R1CS constraints given by A_0, B_0 and $C_0 = I$ faithfully encode the circuit.

C.5.3 Forward algorithm

Given a column vector $W : \{0,1\}^{\kappa_{\text{block}}} \to E$, the forward algorithm computes the two row-indexed vectors $A_0 W$ and $B_0 W$ without constructing either matrix. The native lincheck verifier uses $W = w_{\text{col}}$ and retains only the scalar $\langle e_{\text{row}},\, A_0 W \rangle + \alpha_{\text{lc}} \langle e_{\text{row}},\, B_0 W \rangle$. The prover of the deferred matrix sumcheck in §10.1.3 instead stores both vectors and folds them through its first κ_{block} rounds.

To compute them, visit the wires from w_1 to w_N and evaluate every expansion against W:

$$\langle \mathcal{F}_w,\, W \rangle = W(k(w)) \;\text{ if } w \text{ is committed}, \qquad \langle \mathcal{F}_w,\, W \rangle = \sum_{u \in S_w} \langle \mathcal{F}_u,\, W \rangle \;\text{ otherwise};$$

then (22) gives

$$\langle A_0(k(w), \cdot),\, W \rangle = \sum_{u \in S_w^A} \langle \mathcal{F}_u,\, W \rangle, \qquad \langle B_0(k(w), \cdot),\, W \rangle = \sum_{u \in S_w^B} \langle \mathcal{F}_u,\, W \rangle. \tag{23}$$

Together with zero at every empty row, these values are the desired vectors.

C.5.4 Transpose algorithm

For $X \in \{A, B\}$, the transpose algorithm computes the complete column-indexed vector $M^X_{\mathrm{col}} = X_0^\top e_{\mathrm{row}}$ without constructing X_0. The prover needs all $2^{\kappa_{\mathrm{block}}}$ entries of both vectors:

$$M^X_{\mathrm{col}}(j) = X_0^\sharp(\upsilon_{\mathrm{zc}}, \chi_{\mathrm{in}}, j) = \sum_{k \in \{0,1\}^{\kappa_{\mathrm{block}}}} X_0(k, j) e_{\mathrm{row}}(k) = (X_0^\top e_{\mathrm{row}})(j), \qquad X \in \{A, B\}, \tag{24}$$

where the second equality uses (18). Viewing each row as a column-indexed vector, empty rows contribute zero, so (22) gives

$$\begin{aligned} M^X_{\mathrm{col}} &= \sum_{w \text{ committed}} e_{\mathrm{row}}(k(w)) \sum_{u \in S^X_w} \mathcal{F}_u \\ &= \sum_u \left(\sum_{\substack{w \text{ committed} \\ u \in S^X_w}} e_{\mathrm{row}}(k(w)) \right) \mathcal{F}_u. \end{aligned}$$

The last equality exchanges the finite sums and collects every occurrence of the same expansion $\mathcal{F}_u$. Thus, for every circuit wire u, define

$$\Gamma^X_u = \sum_{\substack{w \text{ committed} \\ u \in S^X_w}} e_{\mathrm{row}}(k(w)), \qquad \text{so that} \qquad M^X_{\mathrm{col}} = \sum_u \Gamma^X_u \mathcal{F}_u.$$

Run the following procedure independently for $X = A$ and $X = B$. Initialize the output table M^X_{col} to zero and visit $w_N, \ldots, w_1$. At wire w, eliminate the term $\Gamma^X_w \mathcal{F}_w$ using the appropriate case of (21):

- if w is committed, then $\Gamma^X_w \mathcal{F}_w = \Gamma^X_w [\, \cdot = k(w) \,]$, so add Γ^X_w to $M^X_{\mathrm{col}}(k(w))$;
- otherwise, w is an uncommitted $\oplus$ wire and

  $$\Gamma^X_w \mathcal{F}_w = \Gamma^X_w \sum_{u \in S_w} \mathcal{F}_u = \sum_{u \in S_w} \Gamma^X_w \mathcal{F}_u,$$

  so add Γ^X_w to the existing coefficient Γ^X_u of every predecessor $u \in S_w$.

C.6 Soundness analysis

The commitment determines q_{flock}, hence z, before any challenge is drawn, and the PCS proves the claims opened on q_{flock}. So take the one transmitted family to be true of z,

$$s_i = \widetilde{z}(i, \chi'_{\mathrm{in}}, \chi_{\mathrm{out}}), \qquad i \in \{0,1\}^{\kappa_{\mathrm{skip}}},$$

up to the PCS error and the 2^{-160} of Annex A, and let us show a z failing (10) or (11) is rejected.

Lincheck comes first, and it is the only thing that checks anything: the zerocheck's own terminal identity defines v_c instead of testing it, so every lie upstream reaches lincheck as a wrong triple (v_a, v_b, v_c). Its terminal check (17) is evaluated from the true s and the matrices themselves, by the walk of §C.3.1 natively and, in recursion, once (20) is discharged (§10.1.3), so a false (16) survives the 8 rounds with probability at most $16/|E|$ (Fact 3.10, degree two in each round). And by Lemma C.2 and (15), (16) says

$$(v_a + a^\sharp(\upsilon_{\mathrm{zc}}, \chi)) + \alpha_{\mathrm{lc}}(v_b + b^\sharp(\upsilon_{\mathrm{zc}}, \chi)) + \alpha^2_{\mathrm{lc}}(v_c + z^\sharp(\upsilon_{\mathrm{zc}}, \chi)) + \alpha^3_{\mathrm{lc}}(1 + \widetilde{z}(512, \chi_{\mathrm{out}})) = 0,$$

degree three in α_{lc}, all four coefficients fixed before α_{lc} is drawn. Acceptance therefore pins them all, except with $3/|E|$ (Lemma 3.8): v_a, v_b and v_c are the true evaluations, and $\widetilde{z}(512, \chi_{\mathrm{out}}) = 1$, which is (11) except with $\kappa_{\mathrm{batch}}/|E|$ more, χ_{out} being random.

With all three pinned, the zerocheck's terminal identity (13) becomes a real check again: it fixes R_{zc} to the true value of the summand at χ.

Now let z violate (10) at $u = (i, v)$. Then $P(\varphi_8(i))$ is the extension at r of a nonzero Boolean multilinear (Definition C.1, Fact 3.6), and the seven fixed coordinates of r carry $\mathbb{F}_2$-independent weights, so it stays nonzero except with $(\kappa_{\mathrm{flock}} - 7)/|E|$ (Lemma 3.12): the 64 zeros the verifier assumes on $\mathcal{H}$ are false. Let $\hat{P}$ interpolate those zeros together with the received coset values. It vanishes on $\mathcal{H}$ while P does not, so $\hat{P} \neq P$ and $v_P \neq P(v_{\mathrm{zc}})$ except with $127/|E|$ (Corollary 3.9). The κ_{flock} zerocheck rounds then run on a false initial claim with a true terminal value, and reject except with $2\kappa_{\mathrm{flock}}/|E|$.

Summing those terms, such a z is accepted with probability at most

$$\frac{16 + 3 + \kappa_{\mathrm{batch}} + (\kappa_{\mathrm{flock}} - 7) + 127 + 2\kappa_{\mathrm{flock}}}{|E|} = \frac{4\kappa_{\mathrm{batch}} + 163}{|E|} < 2^{-183},$$

on top of the two errors granted above.

D Novel polynomial basis and additive NTT

This annex collects the algebra shared by the PCS (Annex B) and LeanDA (§10.2): additive domains, the novel polynomial basis, its additive FFT (called an NTT over a finite field), and codeword membership. The basis and transform follow [LCH14], as presented in [DP26].

Fix the $\mathbb{F}_2$-basis $e_c := x^c$, $0 \le c < 64$, of K, the first 64 vectors of the basis of E in Annex A. In particular $e_0 = 1$. For a bit vector u, write $\langle u \rangle := \sum_i u_i 2^i$ for its integer index, with bit 0 first.

D.1 Additive domains and subspace polynomials

For $0 \le i \le 64$ let $\mathcal{U}_i := \langle e_0, \ldots, e_{i-1} \rangle_{\mathbb{F}_2}$ and define the *subspace vanishing polynomial* $\mathcal{V}_i(X) := \prod_{u \in \mathcal{U}_i} (X + u)$, of degree 2^i.

Lemma D.1 (Subspace polynomials)**.** *Each $\mathcal{V}_i$ is linearized: it contains only powers X^{2^j}. For $i < 64$: (i) $\mathcal{V}_{i+1}(X) = \mathcal{V}_i(X)(\mathcal{V}_i(X) + \mathcal{V}_i(e_i))$ with $\mathcal{V}_i(e_i) \neq 0$; (ii) $\mathcal{V}_i(X + Y) = \mathcal{V}_i(X) + \mathcal{V}_i(Y)$ as a polynomial identity; in particular the map $\mathcal{V}_i : K \to K$ is $\mathbb{F}_2$-linear.*

Proof. By induction on i; the base case is $\mathcal{V}_0(X) = X$. For (i), split the product defining $\mathcal{V}_{i+1}$ along $\mathcal{U}_{i+1} = \mathcal{U}_i \sqcup (e_i + \mathcal{U}_i)$:

$$\mathcal{V}_{i+1}(X) \;=\; \prod_{u \in \mathcal{U}_i} (X + u) \cdot \prod_{u \in \mathcal{U}_i} (X + e_i + u) \;=\; \mathcal{V}_i(X) \cdot \mathcal{V}_i(X + e_i) \;=\; \mathcal{V}_i(X)(\mathcal{V}_i(X) + \mathcal{V}_i(e_i)),$$

where the middle step recognizes the second product as $\mathcal{V}_i$ evaluated at $X + e_i$, and the last step is the identity (ii) for $\mathcal{V}_i$ with $Y = e_i$. Moreover $\mathcal{V}_i(e_i) \neq 0$, since the roots of $\mathcal{V}_i$ are exactly $\mathcal{U}_i$ and $e_i \notin \mathcal{U}_i$. The recurrence also preserves the linearized form: squaring doubles each exponent, and the other term is a scalar multiple of $\mathcal{V}_i$. For (ii) at $i + 1$, write $\mathcal{V}_{i+1} = \mathcal{V}_i^2 + \mathcal{V}_i(e_i)\,\mathcal{V}_i$ by (i): the second term satisfies the identity by induction, and so does the first, since in characteristic 2

$$\mathcal{V}_i(X + Y)^2 \;=\; \big(\mathcal{V}_i(X) + \mathcal{V}_i(Y)\big)^2 \;=\; \mathcal{V}_i(X)^2 + \mathcal{V}_i(Y)^2. \qquad \square$$

For $i < 64$, let $\widehat{\mathcal{V}}_i := \mathcal{V}_i/\mathcal{V}_i(e_i)$ (the hat denotes normalization, not multilinear extension), so $\widehat{\mathcal{V}}_i(e_i) = 1$ and $\widehat{\mathcal{V}}_0 = X$ (using $e_0 = 1$). By linearity, $\widehat{\mathcal{V}}_i(x) = \sum_c x_c \widehat{\mathcal{V}}_i(e_c)$ for $x = \sum_c x_c e_c$, so given the values $\widehat{\mathcal{V}}_i(e_c)$ (precomputed via the recursion (i)), $\widehat{\mathcal{V}}_i$ is evaluated anywhere in K in at most 64 additions.

D.2 Novel basis and Reed-Solomon encoding

Definition D.2 (Reed-Solomon code)**.** For an additive domain $\mathcal{D} \subseteq K$ of size n and $1 \le k \le n$, let

$$\mathrm{RS}[E, \mathcal{D}, k] := \{(p(x))_{x\in\mathcal{D}} : p \in E[X],\ \deg p < k\}.$$

It has dimension k and rate $\rho = k/n$. Its K-valued subcode is $\mathrm{RS}[K, \mathcal{D}, k]$.

Definition D.3 (Novel basis and encoder [LCH14])**.** Let $0 \le \kappa < 64$, $\nu \ge 0$, and $\kappa + \nu \le 64$. For $0 \le j < 2^\kappa$ with bits j_i, let $\mathcal{X}_j(X) := \prod_i \widehat{\mathcal{V}}_i(X)^{j_i}$; then $\deg \mathcal{X}_j = j$, so $(\mathcal{X}_j)_{j<2^\kappa}$ is a (triangular) basis of the polynomials of degree less than 2^κ. Let $\mathcal{D} := \langle e_0, \dots, e_{\kappa+\nu-1}\rangle_{\mathbb{F}_2}$ and, for a message $g : \{0,1\}^\kappa \to E$,

$$\mathcal{P}_g(X) := \sum_{u\in\{0,1\}^\kappa} g(u) \cdot \mathcal{X}_{\langle u\rangle}(X), \qquad \mathrm{Enc}_{\kappa,\nu}(g) := \big(\mathcal{P}_g(x)\big)_{x\in\mathcal{D}},$$

a linear injection onto $\mathrm{RS}[E, \mathcal{D}, 2^\kappa]$ of rate $\rho = 2^{-\nu}$; the $\mathcal{X}_j$ take their values on $\mathcal{D} \subseteq K$, so a K-valued message gives a K-valued word. (The message is read *directly* as novel-basis coefficients; no conversion is applied.)

The payload consists of polynomial coefficients. Any $k = 2^\kappa$ distinct evaluations determine those coefficients by interpolation, but an evaluation prefix need not equal the payload.

D.3 The additive NTT

The encoder is computed by the additive NTT of [LCH14], a Cooley-Tukey-style butterfly network in which the squaring map is replaced by degree-2 linearized maps ψ_d collapsing the domain, and the twiddles are evaluations of the $\widehat{\mathcal{V}}_d$. For $0 \le d < \kappa$ define

$$\psi_d(X) \ := \ \frac{\mathcal{V}_d(e_d)^2}{\mathcal{V}_{d+1}(e_{d+1})} \cdot X(X+1).$$

Lemma D.4 (Chain structure)**.** *Each ψ_d is $\mathbb{F}_2$-linear with kernel $\{0,1\}$, and $\psi_d \circ \widehat{\mathcal{V}}_d = \widehat{\mathcal{V}}_{d+1}$, hence $\widehat{\mathcal{V}}_d = \psi_{d-1} \circ \cdots \circ \psi_0$. Setting $\mathcal{D}^{(d)} := \widehat{\mathcal{V}}_d(\mathcal{D})$ for $0 \le d \le \kappa$, each $\mathcal{D}^{(d)}$ is an $\mathbb{F}_2$-subspace of dimension $\kappa + \nu - d$. For $d < \kappa$, it contains $1 = \widehat{\mathcal{V}}_d(e_d)$, and ψ_d maps $\mathcal{D}^{(d)}$ onto $\mathcal{D}^{(d+1)}$ two-to-one with fibers $\{x, x+1\}$.*

Proof. $X(X+1) = X^2 + X$ is linear with kernel $\{0,1\}$. By Lemma D.1(i), $\widehat{\mathcal{V}}_{d+1} = \frac{\mathcal{V}_d(\mathcal{V}_d + \mathcal{V}_d(e_d))}{\mathcal{V}_{d+1}(e_{d+1})} = \frac{\mathcal{V}_d(e_d)^2}{\mathcal{V}_{d+1}(e_{d+1})}\widehat{\mathcal{V}}_d(\widehat{\mathcal{V}}_d + 1) = \psi_d \circ \widehat{\mathcal{V}}_d$; the chain follows from $\widehat{\mathcal{V}}_0 = X$. $\widehat{\mathcal{V}}_d|_{\mathcal{D}}$ is linear with kernel $\mathcal{U}_d \subseteq \mathcal{D}$, so its image has dimension $\kappa+\nu-d$, and $\psi_d(\mathcal{D}^{(d)}) = \widehat{\mathcal{V}}_{d+1}(\mathcal{D}) = \mathcal{D}^{(d+1)}$ with $\ker \psi_d = \{0,1\} \subseteq \mathcal{D}^{(d)}$. □

Lemma D.5 (Even-odd refinement)**.** *For $0 \le d \le \kappa$ define $\widehat{\mathcal{V}}_b^{(d)} := \psi_{d+b-1} \circ \cdots \circ \psi_d$ (and $\widehat{\mathcal{V}}_0^{(d)} := X$), and $\mathcal{X}_j^{(d)} := \prod_b (\widehat{\mathcal{V}}_b^{(d)})^{j_b}$, so $\mathcal{X}_j^{(0)} = \mathcal{X}_j$. If $d < \kappa$, $\mathcal{P} = \sum_{j<2^{\kappa-d}} a_j \mathcal{X}_j^{(d)}$, and $\mathcal{P}_\iota := \sum_j a_{2j+\iota} \mathcal{X}_j^{(d+1)}$, then*

$$\mathcal{P}(X) \ = \ \mathcal{P}_0(\psi_d(X)) + X \cdot \mathcal{P}_1(\psi_d(X)).$$

Proof. $\widehat{\mathcal{V}}^{(d)}_{b+1} = \widehat{\mathcal{V}}^{(d+1)}_{b} \circ \psi_d$, so $\mathcal{X}^{(d)}_{2j} = \mathcal{X}^{(d+1)}_{j} \circ \psi_d$ and $\mathcal{X}^{(d)}_{2j+1} = X \cdot \mathcal{X}^{(d)}_{2j}$; split the sum by parity. □

Additive NTT [LCH14]. *Input:* coefficients $a_0, \ldots, a_{2^\kappa - 1}$ of $\mathcal{P} = \sum_j a_j \mathcal{X}_j$. *Output:* $(\mathcal{P}(x))_{x \in \mathcal{D}}$, identifying $\{0,1\}^{\kappa+\nu} \cong \mathcal{D}$ via $v \mapsto x_v := \sum_c v_c e_c$.

1. Initialize $\mathsf{B}(v) := a_{\langle (v_0, \ldots, v_{\kappa-1}) \rangle}$ for all $v \in \{0,1\}^{\kappa+\nu}$ (the coefficient array, tiled 2^ν times).
2. For $d = \kappa - 1$ down to 0: for every v with $v_d = 0$, with $v' := v \oplus \mathbf{1}_d$ and twiddle $t := \sum_{c>d} v_c \widehat{\mathcal{V}}_d(e_c)$: $\mathsf{B}(v) \mathrel{+}= t \cdot \mathsf{B}(v')$; then $\mathsf{B}(v') \mathrel{+}= \mathsf{B}(v)$.
3. Output B; then $\mathsf{B}(v) = \mathcal{P}(x_v)$.

Proposition D.6 (Correctness and cost)**.** *The algorithm computes* $\mathrm{Enc}_{\kappa,\nu}$ *with exactly* $\frac{1}{2}\kappa n$ *multiplications and* κn *additions in the butterflies, plus fewer than* n *additions for the twiddles:* t *depends only on* $(v_c)_{c>d}$*, so stage* d *has* $2^{\kappa+\nu-1-d}$ *distinct twiddles, shared by the* 2^d *butterflies of a block and enumerable with one addition each,* $\sum_{d<\kappa} 2^{\kappa+\nu-1-d} < n$ *in total.*

Proof sketch. For $\ell \in \{0,1\}^d$ let $\mathcal{P}^{(d)}_\ell := \sum_{j < 2^{\kappa-d}} a_{\langle \ell \rangle + 2^d j} \mathcal{X}^{(d)}_j$ collect the coefficients congruent to $\langle \ell \rangle$ modulo 2^d. Downward induction on d maintains the invariant that after stages $\kappa - 1, \ldots, d$, $\mathsf{B}(v) = \mathcal{P}^{(d)}_{(v_0,\ldots,v_{d-1})}(\widehat{\mathcal{V}}_d(x_v))$, where by linearity $\widehat{\mathcal{V}}_d(x_v) = v_d + t$ with t the twiddle of the algorithm (coordinates $c < d$ lie in $\ker \widehat{\mathcal{V}}_d = \mathcal{U}_d$, and $\widehat{\mathcal{V}}_d(e_d) = 1$). The base $d = \kappa$ is the tiled initialization. For the step, let $\ell := (v_0, \ldots, v_{d-1})$: the order-$(d{+}1)$ even and odd parts of $\mathcal{P}^{(d)}_\ell$ are $\mathcal{P}^{(d+1)}_{(\ell,0)}$ and $\mathcal{P}^{(d+1)}_{(\ell,1)}$ (as $\langle \ell \rangle + 2^d(2j + \iota) = \langle (\ell, \iota) \rangle + 2^{d+1} j$), whose values at $\widehat{\mathcal{V}}_{d+1}(x_v) = \widehat{\mathcal{V}}_{d+1}(x_{v'})$ are the pre-stage entries $\mathsf{B}(v)$ and $\mathsf{B}(v')$; Lemma D.5 at the fiber $\{t, t+1\}$ of ψ_d over this point (Lemma D.4) gives $\mathcal{P}^{(d)}_\ell(t) = \mathsf{B}(v) + t\,\mathsf{B}(v')$ and $\mathcal{P}^{(d)}_\ell(t+1) = \mathcal{P}^{(d)}_\ell(t) + \mathsf{B}(v')$, exactly the butterfly. At $d = 0$, $\mathsf{B}(v) = \mathcal{P}(x_v)$. Each of the κ stages does $n/2$ butterflies of one multiplication and two additions. □

D.4 Column weights

A codeword coordinate is a linear combination of the message coefficients. The following factorization lets the PCS verifier evaluate that combination's multilinear weight without running a transform.

Lemma D.7 (Column weights are tensors)**.** *For* $x \in \mathcal{D}$ *let* $W_x(u) := \prod_i \widehat{\mathcal{V}}_i(x)^{u_i}$*. Then* $\mathrm{Enc}_{\kappa,\nu}(g)[x] = \langle W_x, g \rangle := \sum_u W_x(u) g(u)$*, and*

$$\widetilde{W_x}(r) \;=\; \prod_{i=0}^{\kappa-1} \big((1 + r_i) + r_i \widehat{\mathcal{V}}_i(x)\big),$$

so one evaluation of $\widetilde{W_x}$ *costs* $O(\kappa)$ *multiplications and additions in total: the values* $\widehat{\mathcal{V}}_i(x)$ *follow the chain of Lemma D.4,* $\widehat{\mathcal{V}}_0(x) = x$ *and* $\widehat{\mathcal{V}}_{i+1}(x) = \psi_i(\widehat{\mathcal{V}}_i(x))$*, at two multiplications and one addition per step (the constants of the* ψ_i *are precomputed once, shared by all levels and queries), and the* κ*-factor product costs the same again.*

Proof. The first identity is Definition D.3 expanded: $\mathcal{P}_g(x) = \sum_u g(u) \prod_i \widehat{\mathcal{V}}_i(x)^{u_i} = \langle W_x, g \rangle$. For the second, start from the definition of the multilinear extension and factor eq coordinate-wise:

$$\widetilde{W_x}(r) \;=\; \sum_{u \in \{0,1\}^\kappa} \mathrm{eq}(u, r)\, W_x(u) \;=\; \sum_{u \in \{0,1\}^\kappa} \prod_{i=0}^{\kappa-1} \mathrm{eq}(u_i, r_i)\, \widehat{\mathcal{V}}_i(x)^{u_i}.$$

Each factor of the summand depends on a single bit u_i, so by distributivity the sum over $\{0,1\}^\kappa$ factors into a product of one-bit sums:

$$\widetilde{W_x}(r) \;=\; \prod_{i=0}^{\kappa-1} \sum_{u_i\in\{0,1\}} \mathrm{eq}(u_i, r_i)\, \widehat{\mathcal{V}}_i(x)^{u_i} \;=\; \prod_{i=0}^{\kappa-1} ((1+r_i)\cdot 1 + r_i \cdot \widehat{\mathcal{V}}_i(x)),$$

using $\mathrm{eq}(0, r_i) = 1 + r_i$ and $\mathrm{eq}(1, r_i) = r_i$. □

D.5 Duality and codeword membership

Lemma D.8 (Summation on an additive domain)**.** *Let $\mathcal{D} = \mathcal{U}_{\log_2 n}$, with $n \geq 2$ a power of two. The derivative $\mathcal{V}'_{\log_2 n}$ is a nonzero constant. For every polynomial h of degree less than n,*

$$\sum_{x\in\mathcal{D}} h(x) = \mathcal{V}'_{\log_2 n}\,[X^{n-1}]\,h, \tag{25}$$

where $[X^{n-1}]\,h$ denotes the coefficient of X^{n-1} in h. In particular, the sum is zero if $\deg h \leq n-2$.

Proof. The vanishing polynomial $\mathcal{V}_{\log_2 n}(X) = \prod_{x\in\mathcal{D}}(X - x)$ is linearized (Lemma D.1): its only powers of X are $X, X^2, X^4, \ldots$. In characteristic two, the derivative of every term except the X term is zero. Its derivative is therefore constant, equal to the coefficient of X. This constant is nonzero: at any root $x \in \mathcal{D}$, differentiating the product gives $\mathcal{V}'_{\log_2 n}(x) = \prod_{y\in\mathcal{D}\setminus\{x\}}(x - y)$, a product of nonzero factors.

Now let h have degree less than n. Its values at the n domain points determine it by Lagrange interpolation:

$$h(X) = \sum_{x\in\mathcal{D}} \frac{h(x)}{\mathcal{V}'_{\log_2 n}(x)} \frac{\mathcal{V}_{\log_2 n}(X)}{X - x}.$$

Each quotient $\mathcal{V}_{\log_2 n}(X)/(X - x)$ is monic of degree $n - 1$, so its coefficient of X^{n-1} is 1. Taking that coefficient on both sides, and using the fact that the derivative is the same constant at every x, gives

$$[X^{n-1}]\,h = \sum_{x\in\mathcal{D}} \frac{h(x)}{\mathcal{V}'_{\log_2 n}(x)} = \frac{1}{\mathcal{V}'_{\log_2 n}} \sum_{x\in\mathcal{D}} h(x).$$

Multiplying by the nonzero derivative gives (25). □

Corollary D.9 (Dual code)**.** *For $1 \leq k < n$ on the domain of Lemma D.8,*

$$\mathrm{RS}[E, \mathcal{D}, k]^\perp = \mathrm{RS}[E, \mathcal{D}, n - k].$$

At rate $1/2$, the code equals its dual.

Proof. If f and g have degrees less than k and $n - k$, then $\deg(fg) \leq n - 2$. Lemma D.8 gives $\sum_{x\in\mathcal{D}} f(x)g(x) = 0$, so the two codes are orthogonal. Their dimensions sum to n, giving equality with the dual. □

Proposition D.10 (Membership with tensor weights)**.** *Let $k = 2^\kappa$, $n = 2k$, and $\mathcal{D} = \mathcal{U}_{\kappa+1}$. For $z = (z_0, \ldots, z_{\kappa-1}) \in E^\kappa$, define*

$$L(X) := \prod_{j<\kappa} (1 + z_j \widehat{\mathcal{V}}_j(X)). \tag{26}$$

Its evaluation vector is a codeword. For any fixed $w \in E^{\mathcal{D}}$, the inner product $\langle L, w\rangle := \sum_{x\in\mathcal{D}} L(x)w(x)$ vanishes identically as a polynomial in z if and only if w is a codeword. If w is not a codeword and the z_j are independent and uniform, then

$$\Pr_z[\langle L, w\rangle = 0] \le \frac{\kappa}{|E|}.$$

Proof. Since $\widehat{\mathcal{V}}_j = \mathcal{X}_{2^j}$, expanding (26) gives the novel-basis polynomials of index less than k, with coefficient tensor $\bigotimes_j(1, z_j)$. Thus $\deg L < k$, and Corollary D.9 makes it orthogonal to every codeword.

Conversely, the coefficient of $\prod_j z_j^{u_j}$ in $\langle L, w\rangle$ is $\langle \mathcal{X}_{\langle u\rangle}, w\rangle$. If the inner product vanished identically, w would be orthogonal to every basis codeword, hence would itself belong to the code. For an invalid w, it is therefore a nonzero multilinear polynomial of total degree at most κ. The probability bound follows from Lemma 3.8. □

References

[ACFY25] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. WHIR: Reed–solomon proximity testing with super-fast verification. In *Advances in Cryptology – EUROCRYPT 2025*, 2025. Cryptology ePrint Archive, Paper 2024/1586. URL: https://eprint.iacr.org/2024/1586.

[BEG+94] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2–3):225–244, 1994.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ signature framework. In *ACM Conference on Computer and Communications Security (CCS 2019)*, 2019. URL: https://sphincs.org/.

[BRW26] Benedikt Bünz, Ron Rothblum, and William Wang. Flock: Fast proving for batch boolean computations. Cryptology ePrint Archive, Paper 2026/1329, 2026. URL: https://eprint.iacr.org/2026/1329.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018. URL: https://eprint.iacr.org/2018/046.

[BSCH+25] Eli Ben-Sasson, Dan Carmon, Ulrich Haböck, Swastik Kopparty, and Shubhangi Saraf. On proximity gaps for reed–solomon codes. Cryptology ePrint Archive, Paper 2025/2055, 2025. URL: https://eprint.iacr.org/2025/2055.

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Theory of Cryptography (TCC 2016-B)*, 2016.

[CCH+19] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat–shamir: From practice to theory. In *Symposium on Theory of Computing (STOC 2019)*, 2019.

[DP23] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023. URL: https://eprint.iacr.org/2023/1784, doi:10.1007/978-3-031-91134-7_4.

[DP26] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. In *Advances in Cryptology – EUROCRYPT 2026*, pages 3–32. Springer, 2026. Cryptology ePrint Archive, Paper 2024/504. URL: https://eprint.iacr.org/2024/504, doi:10.1007/978-3-032-25336-1_1.

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

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo: a turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021. URL: https://eprint.iacr.org/2021/1063.

[Gru24] Angus Gruen. Some improvements for the piop for zerocheck. Cryptology ePrint Archive, Paper 2024/108, 2024. URL: https://eprint.iacr.org/2024/108.

[HBG+18] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, 2018. URL: https://datatracker.ietf.org/doc/html/rfc8391.

[HJR+25] Tamir Hemo, Kevin Jue, Eugene Rabinovich, Gyumin Roh, and Ron D. Rothblum. Jagged polynomial commitments (or: How to stack multilinears). Cryptology ePrint Archive, Paper 2025/917, 2025. URL: https://eprint.iacr.org/2025/917.

[Irr] Irreducible. Multi-multiset matching (M3). https://www.binius.xyz/basics/binius-v0/m3/. Binius documentation.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to reed–solomon erasure codes. In *Foundations of Computer Science (FOCS 2014)*, 2014.

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4), 1992.

[MWKR26] Long Meng, Benedikt Wagner, George Kadianakis, and Francesco Risitano. LeanDA: Design and benchmark. Ethereum Research, 2026. URL: https://ethresear.ch/t/leanda-design-and-benchmark/25642.

[NA25] Andrija Novakovic and Guillermo Angeris. Ligerito: A small and concretely fast polynomial commitment scheme. Cryptology ePrint Archive, Paper 2025/1187, 2025. URL: https://eprint.iacr.org/2025/1187.

[SA15] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The blake2 cryptographic hash and message authentication code (mac). RFC 7693, Internet Engineering Task Force, 2015. URL: https://www.rfc-editor.org/rfc/rfc7693, doi:10.17487/RFC7693.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[Wag26] Benedikt Wagner. Formally verified security for PQ-DAS / leanDA. Ethereum Research, August 2026. URL: https://ethresear.ch/t/formally-verified-security-for-pq-das-leanda/25746.