Whirlaway: multilinear PIOP for AIR

Thomas Coratger, Tom Wambsgans

1 Introduction

AIR (Algebraic Intermediate Representation) is a common arithmetization standard is the context of (ZK)-SNARK (Succint Non-Interactive Argument of Knowledge, potentially Zero-Knowledge). Traditionally, AIR constraints were proven using a univariate PIOP (Polynomial Interactive Oracle Proof), as explained in the EthStark paper [1]. Whirlaway is a simple multilinear PIOP, proving AIR constraints, focusing on lightweight proofs.

The multilinear framework has two main advantages compared to classical approaches. First, it enables encoding and committing the entire AIR execution trace as a single multilinear polynomial, instead of encoding each table column as a separate low-degree univariate polynomial. In many modern proof systems, the correctness of a computation is expressed through multiple tables, each capturing specific CPU instructions (in the context of zkVMs) or constraints. To enable efficient verification, the prover encodes the columns of these tables as low-degree polynomials and commits to them using a polynomial commitment scheme. However, this design introduces a caveat: the prover must commit to and provide evaluation proofs for each polynomial separately, and the verifier must process and check each of these commitments individually. As the number of columns grows, the verifier's workload and the overall proof size scale linearly, which can significantly impact efficiency.

Second, the multilinear approach allows proving the validity of AIR constraints using the sumcheck protocol, which removes the need for a quotient polynomial. In the classical univariate approach, AIR constraints are bundled into a single vanishing condition, and the prover must construct and commit to a quotient polynomial that encodes the division of the constraint polynomial by the vanishing polynomial of the domain. The opening proof for the committed quotient polynomial again increases the proof size. From the prover perspective, using a quotient polynomial coupled with a FRI based PCS (superlinear time) may also be slower than the sumcheck (linear time) algorithm

Paired with the recent WHIR Polynomial Commitment Scheme [2], we obtain a hash-based SNARK, plausibly post quantum, with small proof size (on the order of 128 KiB), and promising proving speed (an experimental implementation reached one million Poseidon2 permutations proven per second, on the KoalaBear field, with 128 bits of security, on a RTX 4090).

An initial implementation can be found here.

2 Definitions and notations

2.1 Multilinear framework

2.1.1 Multilinear polynomial

A multilinear polynomial over variables $X_1, \ldots, X_k$ is a multivariate polynomial where each variable appears with degree at most one. Formally, it can be written as:

$$P(X_1, \ldots, X_k) = \sum_{S \subseteq [k]} c_S \prod_{i \in S} X_i,$$

where each $c_S \in \mathbb{F}_p$ is a coefficient, and the sum ranges over all subsets S of $1, \ldots, k$. For example, over two variables X, Y, the polynomial $3 + 2X + 5Y + 7XY$ is multilinear, but X^2 or Y^3 are not.

A multilinear polynomial is uniquely identified by its evaluation on the boolean hypercube:

$$P(X_1, \ldots, X_k) = \sum_{b \in \{0,1\}^k} eq((X_1, \ldots, X_k), b) \cdot P(b)$$

2.1.2 Multilinear extension

Given a function $f : \{0,1\}^n \to \mathbb{F}_p$, its multilinear extension $\widehat{f} : \mathbb{F}_p^n \to \mathbb{F}_p$ is the unique multilinear polynomial over n variables that agrees with f on all Boolean inputs. Explicitly, it can be written as:

$$\widehat{f}(x) = \sum_{b \in \{0,1\}^n} f(b) \cdot eq(b, x)$$

This extension allows evaluating the function not just at Boolean points, but at any point $x \in \mathbb{F}_p^n$.

2.2 Notations

- *log* is always in base 2
- $[i]_2$: big-endian bit decomposition of an integer i
- $eq(x,y) := \prod_{i=1}^{n}(x_i y_i + (1-x_i)(1-y_i))$, for x and y in $\mathbb{F}^n$. This "equality multilinear polynomial" verifies: $eq(x,y) = 1$ if $x = y$, 0 otherwise, for x and y both sampled on the hypercube $\{0,1\}^n$.
- $\mathbb{F}_p$: base field, typically KoalaBear ($p = 2^{31} - 2^{24} + 1$), or BabyBear ($p = 2^{31} - 2^{27} + 1$)
- $\mathbb{F}_q$: extension field ($q = p^\kappa$)
- M (resp. M'): number of columns (resp. non-preprocessed columns) in the AIR table
- m (resp. m'): smallest integer such that $2^m \geq M$ (resp. $2^{m'} \geq M'$)
- $N = 2^n$: number of rows in the AIR table
- $h_1, \ldots, h_u$: transition constraints
- H: batched constraint ($H := \sum_{i=0}^{u-1} h_i \alpha^i$)
- T: multilinear polynomial in $\mathbb{F}_p$ encoding all the (non-preprocessed) columns, with $n + m'$ variables

3 AIR Arithmetization

3.1 Description

In the AIR arithmetization, the witness consists of a list of M columns $c_0, \ldots, c_{M-1}$ (forming the AIR table). Each column contains $N = 2^n$ elements in $\mathbb{F}_p$ (without loss of generality, we use a power-of-two domain). The goal of the prover is to convince the verifier that the table respects a set of u transition constraints $h_0, \ldots, h_{u-1}$. Each constraint h is a polynomial in $2M$ variables, which is respected if for all rows $r \in \{0, \ldots, N-2\}$:

$$h(c_0[r], \ldots, c_{M-1}[r], c_0[r+1], \ldots, c_{M-1}[r+1]) = 0$$

3.2 Preprocessed columns

Traditional AIR systems allow the verifier to fix certain cells in the table (see "boundary conditions" here). For technical reasons, we use a slightly different approach: we allow the verifier to fix certain columns, potentially sparse (called "preprocessed columns"). The work of the verifier associated to each preprocessed column is proportional to its number of nonzero rows. We denote by $c_0, \ldots, c_{M'-1}$ the non-preprocessed columns and $c_{M'}, \ldots, c_{M-1}$ the preprocessed ones.

3.3 Example: Fibonacci sequence

In general, each constraint h_i is a polynomial relation over the current and next rows of the table. We use the notation X_j^{up} to refer to the value of column c_j at row r (the "upper" row), and X_j^{down} to refer to its value at row $r+1$ (the "lower" row). These variables appear as inputs to the h_i constraints, which together enforce the correct behavior across the table. Each constraint has the form:

$$h_i(\underbrace{X_0^{\text{up}}, \ldots, X_{M-1}^{\text{up}}}_{\text{Upper row columns}}, \underbrace{X_0^{\text{down}}, \ldots, X_{M-1}^{\text{down}}}_{\text{Lower row columns}}) = \text{some constraint}$$

where:

- $X_j^{\text{up}} = c_j[r]$ is the value of column c_j at row r
- $X_j^{\text{down}} = c_j[r+1]$ is the value of column c_j at row $r+1$

Let's say the prover wants to convince the verifier that the N-th values of the Fibonacci sequence equals F_N. We use $M = 4$ columns, as illustrated in Figure 1:

- The first $M' = 2$ columns c_0 and c_1 contain the values of the Fibonacci sequence, which is guaranteed by the constraints:
  - h_0 constraint ensures that the next c_1 value equals the sum of the current c_0 and c_1 values, i.e., $F_{r+2} = F_r + F_{r+1}$:

$$h_0(X_0^{\text{up}}, X_1^{\text{up}}, -, -, -, X_1^{\text{down}}, -, -) = X_1^{\text{down}} - (X_0^{\text{up}} + X_1^{\text{up}})$$

  - h_1 constraint shifts the sequence forward, ensuring that the next c_0 value equals the current c_1 value:

$$h_1(-, X_1^{\text{up}}, -, -, X_0^{\text{down}}, -, -, -) = X_0^{\text{down}} - X_1^{\text{up}}$$

- The last two columns c_2 and c_3 are "preprocessed": their content is enforced by the verifier. In our case we set $c_2 = [1, 0, \ldots, 0]$ to act as a selector for the initial row and $c_3 = [0, \ldots, 0, 1]$ as a selector for the final row. We finally use the following constraints, to ensure that the 2 initial values of the sequence are correct (0 and 1), and that the final value equals F_N:
  - h_2 ensures that $c_0 = 0$ when $c_2 \neq 0$ (which only occurs at the first row):

$$h_2(X_0^{\text{up}}, -, X_2^{\text{up}}, -, -, -, -, -) = X_2^{\text{up}} \cdot X_0^{\text{up}}$$

  - h_3 ensures $c_1 = 1$ when $c_2 \neq 0$ (only at the first row).

$$h_3(-, X_1^{\text{up}}, X_2^{\text{up}}, -, -, -, -, -) = X_2^{\text{up}} \cdot (X_1^{\text{up}} - 1)$$

  - h_4 ensures $c_0 = F_N$ when $c_3 \neq 0$ (only at the last row).

$$h_4(X_0^{\text{up}}, -, -, X_3^{\text{up}}, -, -, -, -) = X_3^{\text{up}} \cdot (X_0^{\text{up}} - F_n)$$

  Note that c_2 and c_3 are sparse, both contain only one non-zero index. As a consequence, they have a negligible impact on the verification time.

Row	c_0	c_1	c_2 (preproc.)	c_3 (preproc.)
0	0	1	1	0
1	1	1	0	0
2	1	2	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
N-1	F_N	F_{N+1}	0	1

Figure 1: Fibonacci sequence AIR table layout with preprocessed selectors

4 Proving system

4.1 Commitment

Contrary to most of the STARK systems, which use a univariate Polynomial Commitment Scheme (PCS), like FRI or KZG, we use instead a multilinear one.

The entire AIR table is encoded and committed as a single multilinear polynomial T (except for the preprocessed columns, which are not committed). T has $n + m'$ variables, where $n = \log N = \log(\text{number of rows})$ and $m' = \lceil \log M' \rceil = \lceil \log(\text{number of non-preprocessed columns}) \rceil$. T is defined by its evaluations over the boolean hypercube: for every (non-preprocessed) column index i ($0 \le i < M'$) and for every row index r ($0 \le r < N$):

$$T([i]_2[r]_2) := c_i[r],$$

where:

- $[i]_2$ is the big-endian bit decomposition of i into m' bits (where $m' = \lceil \log M' \rceil$)
- $[r]_2$ is the big-endian bit decomposition of r into n bits (where $n = \log N$)
- $[i]_2[r]_2$ represents concatenated $m' + n$ bits

For example, if $M' = 20$, $N = 128$, $i = 3$, and $r = 33$, we have:

$$[i]_2 = 00011, \quad [r]_2 = 0100001, \quad [i]_2[r]_2 = 00011 \,|\, 0100001.$$

This means that T evaluates at the point corresponding to these bits to give:

$$T(00011 \,|\, 0100001) = c_3[33]$$

The undefined evaluations (for indices $M' \le i < 2^{m'}$) are irrelevant and can be set to zero.

Note that the coefficients of T are in the base field $\mathbb{F}_p$, while the verifier will later query T at a random evaluation point drawn from the extension field $\mathbb{F}_q$. Querying over the larger extension field is important for soundness because it reduces the prover's chance of successfully cheating: the Schwartz-Zippel lemma guarantees that a nonzero polynomial will evaluate to zero at a random point with probability at most $\deg / |\mathbb{F}_q|$, and using a larger field strengthens this bound.

One solution is to embed T into the extension field $\mathbb{F}_q$ before committing to it, but this create overhead. Section 6 describes a simple and efficient approach to avoid this "embedding overhead" with WHIR.

4.2 Batching the constraints

After receiving the commitment to T, the verifier sends a random scalar $\alpha \in \mathbb{F}_q$ to the prover. Up to a small soundness error, we can replace the u transition constraints by a single one:

$$H := \sum_{i=0}^{u-1} h_i \alpha^i$$

The batched constraint H is a single combined polynomial that aggregates all the individual transition constraints $h_0, \ldots, h_{u-1}$. Instead of checking each h_i separately, we check H once, which compresses the verification work while preserving soundness.

4.3 Zerocheck

The main argument comes from [3] (see also [4]). For each column c, we define two multilinear polynomials over n variables:

- The shifted copy c^{up},
- The forward-shifted copy c^{down}.

Specifically, as illustrated in Figure 2:

$$\begin{aligned} c^{\text{up}}([r]_2) &= \begin{cases} c[r] & \text{if } r \in \{0, \dots, N-2\} \\ c[N-2] & \text{if } r = N-1 \end{cases} \\ c^{\text{down}}([r]_2) &= \begin{cases} c[r+1] & \text{if } r \in \{0, \dots, N-2\} \\ c[N-1] & \text{if } r = N-1 \end{cases} \end{aligned} \tag{1}$$

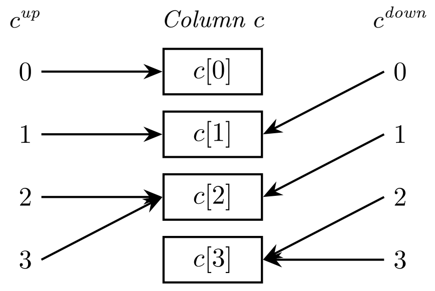

Figure 2: Visualization of c^{up} and c^{down} for $N = 4$.

The batched constraint H is respected on the table if and only if:

$$\forall r \in \{0, \dots, N-2\}, \quad H(c_0[r], \dots, c_{M-1}[r], c_0[r+1], \dots, c_{M-1}[r+1]) = 0$$
$$\Leftrightarrow$$
$$\forall r \in \{0, \dots, N-1\}, \quad H(c_0^{\text{up}}([r]_2), \dots, c_{M-1}^{\text{up}}([r]_2), c_0^{\text{down}}([r]_2), \dots, c_{M-1}^{\text{down}}([r]_2)) = 0$$

The last equality can be proven using a zerocheck (see [5]), assuming the verifier has oracle access to $c_0^{\text{up}}, \dots, c_{M-1}^{\text{up}}$ and $c_0^{\text{down}}, \dots, c_{M-1}^{\text{down}}$, which will be addressed in Section 4.4. The zerocheck is performed as follows:

1. The verifier sends a random vector $r \in \mathbb{F}_q^n$,
2. Prover and verifier run the sumcheck protocol to prove that:

$$\sum_{b \in \{0,1\}^n} eq(b, r) \cdot H(c_0^{\text{up}}(b), \dots, c_{M-1}^{\text{up}}(b), c_0^{\text{down}}(b), \dots, c_{M-1}^{\text{down}}(b)) = 0$$

3. At each round i of the sumcheck, the verifier sends a random challenge $\beta_i \in \mathbb{F}_q$. At the end of the final round, the verifier needs to evaluate the expression inside the sum above for $b \leftarrow \beta = (\beta_1, \dots, \beta_n)$.
   - The factor $eq(\beta, r)$ can be computed directly by the verifier.
   - For the remaining part, the prover provides the claimed evaluations (how the verifier checks the correctness of these values will be detailed in Section 4.4.):

$$c_0^{\text{up}}(\beta), \dots, c_{M-1}^{\text{up}}(\beta), \quad c_0^{\text{down}}(\beta), \dots, c_{M-1}^{\text{down}}(\beta)$$

   Given these $2M$ values, the verifier can finally evaluate H, which concludes the zerocheck.

4.4 Oracle access to c^{up} and c^{down}

In Section 4.3, for each column c_i, the prover has sent two values: $\text{claim}_i^{\text{up}}$ and $\text{claim}_i^{\text{down}}$ respectively equal to $c_i^{\text{up}}(\beta)$ and $c_i^{\text{down}}(\beta)$ in the honest case. It is now time to prove the correctness of these $2M$ evaluations.

First, the verifier sends a random challenge $\gamma \in \mathbb{F}_q$. Except with small soundness error, the $2M$ claims can be reduced to the following:

$$\sum_{i=0}^{M-1} (\gamma^i \cdot \text{claim}_i^{\text{up}} + \gamma^{i+M} \cdot \text{claim}_i^{\text{down}}) \overset{?}{=} \sum_{i=0}^{M-1} (\gamma^i \cdot c_i^{\text{up}}(\beta) + \gamma^{i+M} \cdot c_i^{\text{down}}(\beta)) \tag{2}$$

- Left-hand side: the verifier can compute it directly using the claimed values.
- Right-hand side: we need explicit formulas for c_i^{up} and c_i^{down} in terms of the multilinear extension of the the corresponding column c_i:

4.4.1 Expression of c^{up}

For every column c, for every $r \in \mathbb{F}_q^n$, we have:

$$\begin{aligned} c^{\text{up}}(r) &= \sum_{b \in \{0,1\}^n} \text{shift}^{\text{up}}(r,b) \cdot \widehat{c}(b) \\ &= \sum_{b \in \{0,1\}^n} \left[\overbrace{eq(b,r) \cdot (1 - eq(r, (\underbrace{1,\dots,1}_{n \text{ times}})))}^{\text{picks } c[r] \text{ when } r \neq N_1 \text{ (i.e. before the last row)}} + \overbrace{eq((r,b),(\underbrace{1,\dots,1}_{2n-1 \text{ times}},0))}^{\text{picks } c[N-2] \text{ when } r = N-1} \right] \cdot \widehat{c}(b), \end{aligned}$$

where:

- $\widehat{c}$ represents the multilinear extension (MLE) of c,
- $\text{shift}^{\text{up}}(r,b)$ is a selector polynomial that determines how each b contributes to the sum.

4.4.2 Expression of c^{down}

For any column c, we define its "down-shifted" version c^{down}. For any point $r \in \mathbb{F}_q^n$, the expression is:

$$\begin{aligned} c^{\text{down}}(r) &= \sum_{b \in 0,1^n} \text{shift}^{\text{down}}(r,b) \cdot \widehat{c}(b) \\ &= \sum_{b \in 0,1^n} \left[\overbrace{\text{next}(r,b)}^{\text{picks } c[r+1] \text{ when } r \neq N-1} + \overbrace{eq((r,b),(\underbrace{1,\dots,1}_{2n \text{ times}}))}^{\text{picks } c[N-1] \text{ when } r = N-1} \right] \cdot \widehat{c}(b), \end{aligned}$$

where:

- $\widehat{c}$ represents the multilinear extension (MLE) of c,
- $\text{shift}^{\text{down}}(r,b)$ is a selector polynomial that determines how each b contributes to the sum,
- "next" is the multilinear polynomial in $2n$ variables defined on the hypercube by:

$$\text{next}([x]_2[y]_2) = \begin{cases} 1 & \text{if } y = x+1 \\ 0 & \text{otherwise} \end{cases} \quad \text{for every pair of n-bit integers } (x,y)$$

See section 5.1 of [3] for more details.

4.4.3 Final sumcheck

The right side of (2) can thus be expressed as:

$$\sum_{b\in\{0,1\}^n} \underbrace{\sum_{i=0}^{M-1} [\gamma^i \cdot \text{shift}^{\text{up}}(\beta, b) + \gamma^{i+M} \cdot \text{shift}^{\text{down}}(\beta, b)] \cdot \widehat{c}_i(b)}_{\text{expr}(\beta,b)}$$

A second sumcheck (with respect to b) is used to compute this sum. Let $\delta \in \mathbb{F}_q^n$ be the corresponding vector of challenges. The verifier must finally evaluate $\text{expr}(\beta, \delta)$. Both shift^{up} and $\text{shift}^{\text{down}}$ can be succinctly computed. It remains $(\widehat{c}_i(\delta))_{0\le i<M}$, that is, the values of the columns' multilinear extensions at the common point δ.

4.5 PCS opening

At the end of the protocol, the verifier needs to check that the prover's claimed evaluations of the multilinear extensions $\widehat{c}_i$ at the point δ are correct. This is done as follows:

1. For the preprocessed columns ($M' \le i < M$), the verifier can directly compute the values $\widehat{c}_i(\delta)$, since the verifier already knows these columns.
2. For the non-preprocessed columns ($0 \le i < M'$), the prover sends the claimed values v_i, which should equal $\widehat{c}_i(\delta)$ if the prover is honest.
3. To efficiently verify all these claims in one go, the verifier samples a random vector $z \in \mathbb{F}_q^{m'}$, where $m' = \lceil \log M' \rceil$. This vector selects a random linear combination over the non-preprocessed columns.
4. Using this z, the verifier computes the combined evaluation:

   $$\sum_{i\in\{0,1\}^{m'}} eq(i, z) \cdot v_i,$$

   which collapses all the prover's claims into a single value. Here, $eq(i, z)$ acts as a selector that matches the random point z.
5. Finally, the verifier requests the prover to open the committed multilinear polynomial T at the combined point (z, δ) and checks that the opening matches the combined evaluation computed above.

This process reduces many separate checks into just one, saving both prover and verifier time, while preserving soundness through the use of random linear combinations.

5 Univariate skip

5.1 The traditional sumcheck protocol

Let's consider the case of the zerocheck (see Section 4.3), which is the most sumcheck-intensive part of the protocol. At this stage, the prover wants to convince the verifier that:

$$\sum_{b\in\{0,1\}^n} eq(b, r) \cdot H(c_0^{\text{up}}(b), \dots, c_{M-1}^{\text{up}}(b), c_0^{\text{down}}(b), \dots, c_{M-1}^{\text{down}}(b)) = 0$$

The traditional sumcheck protocol operates across n rounds, one per variable:

- **Round 1:**

  The prover starts by sending $P_1(X)$, supposedly equal to:

  $$\sum_{b\in\{0,1\}^{n-1}} eq((X, b), r) \cdot H(c_0^{\text{up}}(X, b), \dots, c_{M-1}^{\text{down}}(X, b)),$$

where $H(c_0^{\text{up}}(X, b), \ldots, c_{M-1}^{\text{down}}(X, b)) = H(c_0^{\text{up}}(b), \ldots, c_{M-1}^{\text{up}}(b), c_0^{\text{down}}(b), \ldots, c_{M-1}^{\text{down}}(b))$ for the sake of simplicity. After receiving P_1, the verifier checks that $P_1(0) + P_1(1) = 0$ and responds with a random challenge $\beta_1 \in \mathbb{F}_q$.

- **Round 2:**
  The prover then sends $P_2(X)$, supposedly equal to
  $$\sum_{b \in \{0,1\}^{n-2}} eq((\beta_1, X, b), r) \cdot H(c_0^{\text{up}}(\beta_1, X, b), \ldots, c_{M-1}^{\text{down}}(\beta_1, X, b))$$
  After receiving P_2, the verifier checks that $P_1(\beta_1) = P_2(0) + P_2(1)$ and responds with a random challenge $\beta_2 \in \mathbb{F}_p$.
- Continue for n rounds, until all variables are fixed to random points.
- At the end of the protocol the verifier must check (with $\beta = (\beta_1, \ldots, \beta_n)$):
  $$P_n(\beta_n) = eq(\beta, r) \cdot H(c_0^{\text{up}}(\beta), \ldots, c_{M-1}^{\text{down}}(\beta))$$

5.2 Where the overhead comes from

For soudness reason, if the base field $\mathbb{F}_p$ is too small, the random challenges β must be sampled in an extension field $\mathbb{F}_q$. While the first round of the sumcheck happens entirely inside the cheaper base field $\mathbb{F}_p$, the moment the verifier sends $\beta_1 \in \mathbb{F}_q$, the prover is forced to fold the multilinear polynomials over $\mathbb{F}_q$ —meaning every subsequent round involves more expensive operations in the extension field.

Indeed, after receiving the first challenge $\beta_1 \in \mathbb{F}_q$, the prover must compute the "folded" multilinear polynomials $c_0^{\text{up}}(\beta_1, \cdot), \ldots, c_{M-1}^{\text{down}}(\beta_1, \cdot)$, whose coefficients are now in the extension field $\mathbb{F}_q$. This may significantly slow down the consecutive rounds, compared to the first one.

In summary, from the prover perspective, the first round is computationally less expensive than the next few ones. The *univariate skip* optimization[6] leverages this asymmetry: by reorganizing the domain, we can perform the first k rounds of sumcheck all at once, entirely within the base field $\mathbb{F}_p$. This leads to major efficiency improvements.

5.3 Changing the evaluation domain

The univariate skip optimization (see [6]) improves prover efficiency by cleverly restructuring the evaluation domain. Traditionally, the sumcheck protocol works over the Boolean hypercube $0, 1^n$, handling one variable per round. To skip k variables all at once, we reshape the domain as:

$$D \times \{0, 1\}^{n-k} \tag{3}$$

where $D \subset \mathbb{F}_p$ is an arbitrary subset of size 2^k— for example, $D = 0, \ldots, 2^k - 1$. This effectively replaces the first k boolean variables with a single univariate variable over D, while keeping the remaining $n - k$ variables in the usual boolean space.

With this new perspective, we define, for $i \in \{0, \ldots, M-1\}$, $\widetilde{c_i}^{\text{up}}(X_1, \ldots, X_{n+1-k})$ as the polynomial, with degree less than 2^k in X_1, and multilinear in the remaining variables, such that:

$$\widetilde{c_i}^{\text{up}}(x, b_1, \ldots, b_{n-k}) = c_i^{\text{up}}([x]_2, b_1, \ldots, b_{n-k}),$$

for all $x \in D$ and $(b_1, \ldots, b_{n-k}) \in \{0, 1\}^{n-k}$, where $[x]_2$ represents the big-endian decomposition of x in k bits.

We define $\widetilde{c_i}^{\text{down}}$ similarly.

Finally, we define $\widetilde{eq}(X_1, \ldots, X_{n+1-k}, Y_1, \ldots, Y_{n+1-k})$ as the polynomial, with degree $< 2^k$ in X_1 and Y_1, and multilinear in the remaining variables, such that:

$$\widetilde{eq}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

for all $(x, y) \in (D \times 0, 1^{n-k})^2$.

5.4 Zerocheck in the new domain

To apply the univariate skip, we need to slightly adapt the zerocheck protocol. Previously, the random challenge r for the zerocheck was drawn from $\mathbb{F}_q^n$, matching the n variables of the hypercube $\{0,1\}^n$. Now, after restructuring the domain to $D \times \{0,1\}^{n-k}$, the challenge must be drawn from the new space:

$$r \in \mathbb{F}_q^{n+1-k},$$

where the first coordinate corresponds to the univariate component $x \in D$, and the remaining coordinates correspond to the $n-k$ boolean variables. With this setup, the zerocheck equation becomes:

$$\sum_{x \in D,\, b \in \{0,1\}^{n-k}} \widetilde{eq}((x,b), r) \cdot H\big(\widetilde{c_0}^{\text{up}}(x,b), \ldots, \widetilde{c}_{M-1}^{\text{up}}(x,b),\, \widetilde{c_0}^{\text{down}}(x,b), \ldots, \widetilde{c}_{M-1}^{\text{down}}(x,b)\big) \overset{?}{=} 0,$$

where:

- $\widetilde{eq}((x,b), r)$ is the extended equality polynomial matching the combined domain.
- H is the batched constraint polynomial, evaluating the sum of all constraints at the selected point.

There is a key difference in the sumcheck protocol. In the traditional hypercube case, after the prover sends the first-round polynomial P_1, the verifier checks:

$$P_1(0) + P_1(1) = 0,$$

but in the new setting, the verifier instead checks:

$$\sum_{x \in D} P_1(x) = 0,$$

where D has size 2^k. This adjustment reflects the fact that, thanks to univariate skip, we are treating the first k Boolean variables as one combined univariate block.

Importantly, when $k = 1$, this construction reduces back to the traditional sumcheck on the hypercube —meaning the univariate skip is a strict generalization of the classic protocol.

In summary, by reorganizing the domain and adapting the sumcheck's first step, the univariate skip allows the prover to efficiently collapse multiple rounds into a single, larger but cheaper univariate sum, all while staying in the base field.

5.5 Cost analysis

Let's break down the cost for the prover when using univariate skip. First, consider the degree of the first-round polynomial P_1. Because we collapse k variables at once over the set D (with $|D| = 2^k$), the degree bound becomes:

$$\deg(P_1) = (\deg(H) + 1)(2^k - 1),$$

where $\deg(H)$ is the degree of the batched constraint polynomial.

An important observation is that for all $x \in D$, we already know that $P_1(x) = 0$. These points correspond to the valid domain, so they provide 2^k "free" evaluations that the prover doesn't need to compute explicitly when interpolating P_1.

However, to fully interpolate P_1, the prover still needs to compute the remaining evaluations. For each such point, the prover sums over 2^{n-k} combinations (from the $n-k$ residual variables). Thus, the total number of prover operations to interpolate P_1 is roughly:

$$[(\deg(H) + 1)(2^k - 1) - 2^k + 1] \cdot 2^{n-k} = \deg(H)(2^k - 1) \cdot 2^{n-k}$$

Interestingly, this matches the cost of performing the first k rounds of the traditional sumcheck:

$$\sum_{i=1}^{k} \deg(H) \cdot 2^{n-i}.$$

But with one key difference. In the traditional protocol, rounds $2, 3, \ldots, k$ involve folded polynomials with coefficients in the extension field $\mathbb{F}_q$, which makes them significantly more expensive to compute. In contrast, univariate skip pushes all this work into the first round, but keeps it entirely within the base field $\mathbb{F}_p$, delivering major efficiency gains in practice.

5.6 Rest of the protocol

After finishing the zerocheck, the verifier still needs to verify the claimed values of the shifted polynomials. Specifically, he wants to check the evaluations of:

$$\widetilde{c_0}^{\text{up}}, \widetilde{c_1}^{\text{up}}, \ldots, \widetilde{c}_{M-1}^{\text{up}}, \quad \widetilde{c_0}^{\text{down}}, \widetilde{c_1}^{\text{down}}, \ldots, \widetilde{c}_{M-1}^{\text{down}},$$

at a random point:

$$\beta = (\beta_1, \beta_2, \ldots, \beta_{n+1-k}) \in \mathbb{F}_q^{n+1-k}.$$

Let's break down how these are computed. For each column c, we have:

$$\widetilde{c}^{\text{up}}(\beta) = \sum_{x \in D} \widetilde{eq}(x, \beta_1) \cdot c^{\text{up}}([x]_2, \beta_2, \ldots, \beta_{n+1-k}).$$

This expression essentially says:

- For the first coordinate β_1, we match it against $x \in D$ using the equality polynomial $\widetilde{eq}$.
- For each x, we map it back to its k-bit decomposition $[x]_2$ and evaluate the original c^{up}.

We can further rewrite it as:

$$\widetilde{c}^{\text{up}}(\beta) = \sum_{b \in \{0,1\}^k} \Gamma(b, \beta_1) \cdot c^{\text{up}}(b, \beta_2, \ldots, \beta_{n+1-k}),$$

where:

$$\Gamma(b, \beta_1) := \sum_{x \in D} eq(b, [x]_2) \cdot \widetilde{eq}(x, \beta_1).$$

The key point is that Γ can be computed efficiently by the verifier. To prove the correctness of these evaluations, we could:

- Either run another sumcheck to reduce the claims about $\widetilde{c}^{\text{up}}$ back to the original c^{up} (handled as described in Section 4.4),
- Or more directly, expand c^{up} itself (see Section 4.4.1) in terms of the base column c, allowing us to run a single sumcheck over $n + k$ variables.

The same approach applies symmetrically for all the $\widetilde{c}^{\text{down}}$ terms. Finally, all these evaluation checks (for the up- and down-shifted polynomials) can be batched together using a random challenge, so that the prover and verifier only need to perform one combined sumcheck over $n + k$ variables.

6 Avoiding the "embedding overhead" in WHIR

In its simplest form, a polynomial commitment scheme (PCS) operates over a single finite field: the prover commits to a polynomial with coefficients in that field and later opens it at a chosen evaluation point, all within the same arithmetic setting. However, in many modern constructions, the situation is more nuanced. It's common for the committed polynomial to have coefficients in a small base field $\mathbb{F}_p$, while the verifier requests openings at points lying in a larger extension field $\mathbb{F}_q$. This shift introduces what's known as the embedding overhead — the prover must effectively "lift" or embed the polynomial's coefficients into the larger field before opening, which can add significant computational cost.

To address this, prior work introduced the ring-switching protocol (see Section 3 of [7]), which provides an elegant way to sidestep the embedding cost by switching the representation of the polynomial directly into the extension field. While this technique is especially well-suited for binary tower fields, it comes with two main limitations:

1. The extension degree $[\mathbb{F}_q : \mathbb{F}_p]$ must be a power of two, which restricts field choices.
2. It adds complexity to the protocol, both conceptually and in terms of implementation effort.

In this section, we propose a simpler alternative to avoid the embedding overhead specifically for the WHIR PCS construction introduced in [2]. Our approach achieves the same efficiency benefits as ring-switching, but without its rigidity or added protocol complexity — offering a more streamlined and flexible design.

6.1 Ring-Switching costs

To understand the efficiency tradeoffs, let's first break down the prover's costs when using the ring-switching approach on a multilinear polynomial P with v variables over the base field $\mathbb{F}_p$. Under ring-switching, the polynomial P is reinterpreted as a multilinear polynomial P' over the larger extension field $\mathbb{F}_q$, where the number of variables is reduced to $v - d$ (with the extension degree $[\mathbb{F}_q : \mathbb{F}_p] = 2^d$).

We'll also introduce:

- $\rho = 1/2^r$, the initial code rate of the Reed–Solomon (RS) code,
- $f_1, f_2, \ldots$, the folding factors applied at each round (these match the k values defined in Section 2.1.3 of [2]).

In the first round of the protocol, the prover's dominant cost comes from performing 2^{f_1} Fast Fourier Transforms (FFTs), each over a domain of size:

$$2^{v-d+r-f_1}$$

in the extension field $\mathbb{F}_q$.

In the second round, the RS domain size is effectively halved. The prover's work is now dominated by 2^{f_2} FFTs, each over a (smaller) domain of size:

$$2^{v-d+r-1-f_2}$$

again over $\mathbb{F}_q$.

This pattern continues across subsequent rounds: as the folding progresses, the RS domain keeps shrinking, and the prover's cost scales with both the number of FFTs and the size of the domains on which they operate.

6.2 Embedding costs

Without applying the ring-switching protocol, the prover's workload in the first round remains quite similar to the ring-switching case. Specifically, the prover must perform 2^{f_1} FFTs, each over a domain of size 2^{v+r-f_1}, but crucially, these computations take place over the smaller base field $\mathbb{F}_p$. While the FFT domain is 2^d times larger than in the ring-switching setup, the field itself is 2^d times smaller, making the overall computational effort comparable in this initial round.

However, the situation changes significantly in the subsequent rounds. After the first round, the polynomial has been folded, and the prover is now forced to operate over the larger extension field $\mathbb{F}_q$. In the second round, for example, the prover must carry out 2^{f_2} FFTs, each over a domain of size $2^{v+r-1-f_2}$. Importantly, the Reed–Solomon domain in this setting is 2^d times larger compared to what it would be under ring-switching, since the base-to-extension embedding was avoided.

As a result, although the first-round costs are effectively balanced, the subsequent rounds become notably more expensive. This slowdown arises from having to perform FFTs over significantly larger domains in the extension field, eroding the overall efficiency that ring-switching was designed to protect. Over multiple rounds, this cumulative cost becomes a meaningful disadvantage, motivating the search for alternative strategies to bypass the embedding overhead.

6.3 A simple solution

To address the overhead introduced by avoiding ring-switching, we propose a straightforward and effective adjustment to the protocol. Specifically, we suggest increasing the folding factor in the first round to $f_1' = f_1 + d$, where d is the extension degree (i.e., $[\mathbb{F}_q : \mathbb{F}_p] = 2^d$). The folding factors for all subsequent rounds remain unchanged, so $f_2' = f_2$, $f_3' = f_3$, and so on.

Additionally, when transitioning from the first to the second round, we propose adjusting the Reed–Solomon domain reduction: rather than simply halving the domain size as in the traditional approach, the domain should be divided by 2^{d+1}. This adjustment compensates for the lack of ring-switching and aligns the domain size with that of the equivalent protocol using ring-switching.

With this modification, the prover's costs in the first round remain comparable to those under ring-switching. While the prover now performs 2^d times more FFTs, these are carried out over a field that is 2^d times smaller, maintaining a balanced computational load.

$$\text{Prover cost (first round)} \sim 2^d \times \text{FFTs over } \mathbb{F}_p \quad \text{vs.} \quad \text{FFTs over } \mathbb{F}_q.$$

From the second round onward, the prover's workload becomes essentially identical to that of the ring-switching setup: the number of FFTs, the domain sizes, and the field arithmetic all align.

Importantly, this adjustment not only preserves prover efficiency but also ensures that verification costs and proof size remain on par with the ring-switching protocol. At each round, the code rate progression is preserved: it begins at $1/2^r$, then advances to $1/2^{r+f_1-1}$, then to $1/2^{r+f_1+f_2-2}$, and so forth. Furthermore, the size of each Merkle tree leaf remains equivalent. The only notable difference is that, in the first round, ring-switching commits to 2^{f_1} elements in the extension field $\mathbb{F}_q$, while our adjusted approach commits to 2^{f_1+d} elements in the base field $\mathbb{F}_p$. Crucially, both representations occupy the same number of bytes, preserving the compactness of the proof.

References

[1] StarkWare, "ethSTARK documentation," Cryptology ePrint Archive, Paper 2021/582, 2021. [Online]. Available: https://eprint.iacr.org/2021/582

[2] G. Arnon, A. Chiesa, G. Fenzi, and E. Yogev, "WHIR: Reed–solomon proximity testing with super-fast verification," 2024. [Online]. Available: https://eprint.iacr.org/2024/1586

[3] S. Setty, J. Thaler, and R. Wahby, "Customizable constraint systems for succinct arguments," 2023. [Online]. Available: https://eprint.iacr.org/2023/552

[4] W. Borgeaud, "A simple multivariate air argument inspired by superspartan," 2023. [Online]. Available: https://solvable.group/posts/super-air/

[5] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "HyperPlonk: Plonk with linear-time prover and high-degree custom gates," 2022. [Online]. Available: https://eprint.iacr.org/2022/1355

[6] A. Gruen, "Some improvements for the PIOP for ZeroCheck," 2024. [Online]. Available: https://eprint.iacr.org/2024/108

[7] B. E. Diamond and J. Posen, "Polylogarithmic proofs for multilinears over binary towers," 2024. [Online]. Available: https://eprint.iacr.org/2024/504