# Minimal zkVM for Lean Ethereum (draft 0.4.1)

## 1 What is the goal of this zkVM?

Replacing the BLS signature scheme with a Post-Quantum alternative. One approach is to use stateful hash-based signatures, XMSS, as explained in [1], [2] and [3], and to use a hash-based SNARK to handle aggregation. A candidate hash function is Poseidon2 [4].

We want to be able to:

- **Aggregate** XMSS signatures
- **Merge** those aggregate signatures

The latter involves recursively verifying a SNARK. Both tasks mainly require to prove a lot of hashes. A minimal zkVM (inspired by Cairo [5]) is useful as glue to handle all the logic.

Aggregate / Merge can be unified in a single program, which is the only one the zkVM has to prove (see 6.3 for a visual interpretation):

**Algorithm 1** AggregateMerge

**Public input:** **pub_keys** (of size $n$), **bitfield** ($k$ ones, $n-k$ zeros), **msg** (the encoding of the signed message)

**Private input:** $s > 0$, **sub_bitfields** (of size $s$), **aggregate_proofs** (of size $s-1$), **signatures**

▷ Bitfield consistency

Check: **bitfield** $= \bigcup_{i=0}^{s-1}$ **sub_bitfields**[i]

▷ Verify the first $s-1$ sub_bitfields using aggregate_proofs:

**for** $i \leftarrow 0$ to $s-2$ **do**
    inner_public_input $\leftarrow$ (**pub_keys**, **sub_bitfields**[i], **msg**)
    *snark_verify*("AggregateMerge", inner_public_input, **aggregate_proofs**[i])
**end for**

▷ Verify the last sub_bitfields using signatures

$k \leftarrow 0$
**for** $i \leftarrow 0$ to $n-1$ **do**
    **if** **sub_bitfields**[s-1][i] $= 1$ **then**
        *signature_verify*(**msg**, **pub_keys**[i], **signatures**[k])
        $k \leftarrow k+1$
    **end if**
**end for**

## 2 VM specification

### 2.1 Field

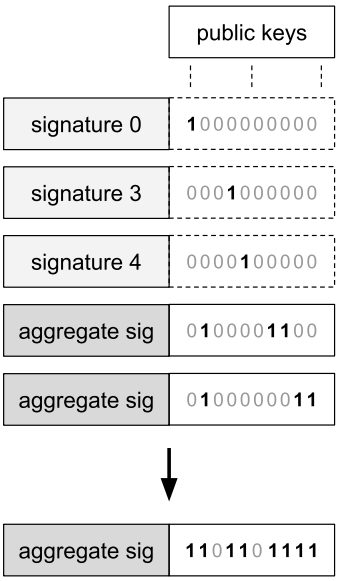
#### 2.1.1 Base field

KoalaBear prime: $p = 2^{31} - 2^{24} + 1$

Advantages:

- small field → less Poseidon rounds

Figure 1: AggregateMerge visualized.




- $x \to x^3$ is an automorphism of $\mathbb{F}_p^*$, meaning efficient S-box for Poseidon2 (in BabyBear, it's degree 7)
- $< 2^{31} \to$ the sum of 2 field elements can be stored in an u32

The small 2-addicity (24) is not a problem in WHIR, thanks to the use of an interleaved Reed Solomon code (and by playing with the folding factors).

#### 2.1.2 Extension field

Extension of dimension 5 or 6

Dimension 5 would require the conjecture 4.12 "up to capacity" in WHIR [6] to get $\approx 128$ bits of security.

### 2.2 Memory

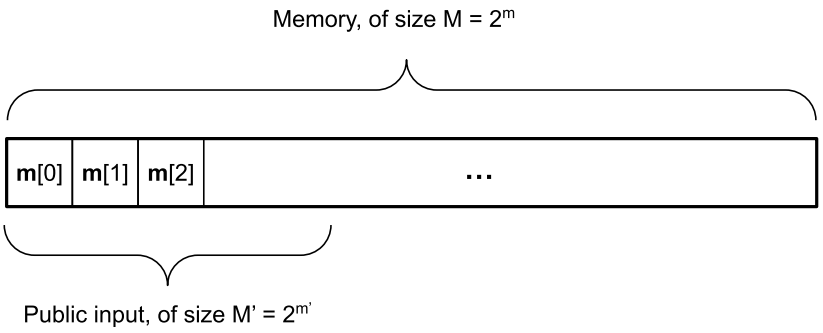
- Read-Only Memory
- Word = KoalaBear field element
- Size: $M = 2^m$ with $16 \leq m \leq 29$. The memory size (which depends on the execution) is communicated at the beginning of the proof.
- The first $M' = 2^{m'}$ memory cells hold the "public input," which is written by the verifier and stores the arguments that the leanISA program may receive as input (in our case: message to sign and XMSS public keys), c.f. Figure 2.

### 2.3 Registers

- pc: program counter
- fp: frame pointer : points to the start of the current stack

**Difference with Cairo: no "ap" register** (allocation pointer).

Figure 2: Memory structure




## 2.4 Instruction Set Architecture

Notations:

- $\alpha$, $\beta$ and $\gamma$ represent parameters of the instructions (immediate value operands)
- $\mathbf{m}[i]$ represents the value of the memory at index $i \in \mathbb{F}_p$, with $i < M$ ($M$: memory size). Any out-of-bound memory access ($i \geq M$) is invalid.
- $\begin{cases} A \\ B \end{cases}$ When using the instruction, either $A$ or $B$ can be used, but not both simultaneously.

### 2.4.1 ADD / MUL

$a + c = b$ or $a \cdot c = b$ with:

$$a = \begin{cases} \alpha \\ \mathbf{m}[\text{fp} + \alpha] \end{cases} \qquad b = \begin{cases} \beta \\ \mathbf{m}[\text{fp} + \beta] \end{cases} \qquad c = \begin{cases} \text{fp} \\ \mathbf{m}[\text{fp} + \gamma] \end{cases}$$

### 2.4.2 DEREF

$$\mathbf{m}[\mathbf{m}[\text{fp} + \alpha] + \beta] = \begin{cases} \gamma \\ \mathbf{m}[\text{fp} + \gamma] \\ \text{fp} \end{cases}$$

### 2.4.3 JUMP (Conditional)

$$\text{condition} = \begin{cases} \alpha \\ \mathbf{m}[\text{fp} + \alpha] \end{cases} \in \{0, 1\} \qquad \text{dest} = \begin{cases} \beta \\ \mathbf{m}[\text{fp} + \beta] \end{cases} \qquad \text{next(fp)} = \begin{cases} \text{fp} \\ \mathbf{m}[\text{fp} + \gamma] \end{cases}$$

$$\text{next(pc)} = \begin{cases} \text{dest} & \text{if condition} = 1 \\ \text{pc} + 1 & \text{if condition} = 0 \end{cases}$$

### 2.4.4 4 Precompiles

1. **POSEIDON_16**: Poseidon2 permutation over 16 field elements.
2. **POSEIDON_24**: Poseidon2 permutation over 24 field elements.
3. **DOT_PRODUCT**: Computes a dot product between two slices of extension field elements. Use cases:

- In WHIR, evaluating the merkle leaves after the initial round (multiple multilinear evaluations in the extension field, but on a common point, so we can compute one time the "Lagrange kernel" and then simply perform dot products.
- A dot product with (1, 1) enables addition in the extension field, with a single instruction.
- A dot product of size one enables multiplication in the extension field, with a single instruction.

4. **MULTILINEAR_EVAL**: interpret a chunk of memory as a multilinear polynomial (in the base field), and evaluates it at a given point (in the extension field.)

   The implementation complexity of this precompile is essentially free: We can simply add "sparse" equality constraints with WHIR. No AIR table needed.

   Use cases:

   - In WHIR, evaluating the merkle leaves at the initial round: $\approx 100$ multilinear evaluations in $\approx 7$ variables.
   - To evaluate the "public memory" of a recursive proof at a random point
   - Potentially, to evaluate the bytecode, for each recursion, at a random point (this is not the only way)

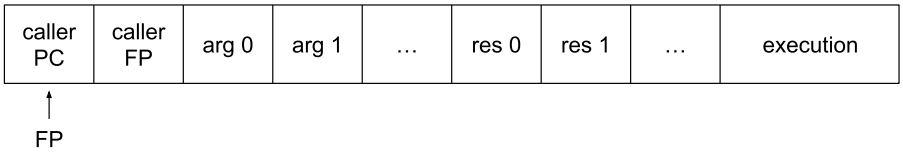
## 2.5 ISA programming

### 2.5.1 Functions

1. Each function has a deterministic memory footprint: the length of the continuous frame in memory that is allocated for each of the its calls.

2. At runtime, each time we call our function, we receive via a memory cell a hint pointing to a free range of memory. We then store the current values of pc / fp at the start of this newly allocated frame, alongside the function's arguments, we can then jump, to enter the function bytecode, and modify fp with the hinted value. The intuition here is that the verifier does not care where the new memory frame will be placed (we use a read-only memory, so we cannot overwrite previous frames). In practice, the prover that runs the program would need to keep the value of the allocation pointer "ap", in order to adequately allocate new memory frames, but there is no need to keep track of it from the versifier's perspective.

Figure 3: Memory layout of a function call




### 2.5.2 Loops

We suggest to unroll loops when the number of iterations is low, and known at compile time. The remaining loops are transformed into recursive functions (by the leanISA compiler).

### 2.5.3 Range checks

It's possible to check that a given memory cell is smaller than some value $t$ (for $t \leq 2^{16}$) in 3 cycles.

We denote by $\mathbf{m}[\text{fp} + x]$ the memory cell for which we want to ensure $\mathbf{m}[\text{fp} + x] < t$. We also denote by $\mathbf{m}[\text{fp} + i]$, $\mathbf{m}[\text{fp} + j]$ and $\mathbf{m}[\text{fp} + k]$ 3 auxiliary memory cells (that have not been used yet).

1. $\mathbf{m}[\mathbf{m}[\text{fp} + x]] = \mathbf{m}[\text{fp} + i]$ (using DEREF, this ensures $\mathbf{m}[\text{fp} + x] < M$, the memory size)

2. $\mathbf{m}[\mathbf{m}[\text{fp} + x]] + \mathbf{m}[\text{fp} + j] = \text{(t-1)}$ (using ADD)

3. $\mathbf{m}[\mathbf{m}[\text{fp} + j]] = \mathbf{m}[\text{fp} + k]$ (using DEREF, this ensures $t - 1 - \mathbf{m}[\text{fp} + x] < M$)

Given $t \leq 2^{16} \leq M$, $\mathbf{m}[\text{fp} + x] < M$, $t - 1 - \mathbf{m}[\text{fp} + x] < M$, and $M \leq 2^{29} < p/2$, we have: $\mathbf{m}[\text{fp} + x] < t$.

Note: From the point of view of the prover running the program, some hints are necessary (filling the values of $\mathbf{m}[\text{fp} + i]$ and $\mathbf{m}[\text{fp} + k]$ must be done at end of execution).

This idea was pointed out by D. Khovratovich, and is an unplanned use of the DEREF instruction.

**Example 2.1.** Let's say we want to write a function with 2 arguments $x = \mathbf{m}[\text{fp} + 2]$ and $y = \mathbf{m}[\text{fp} + 3]$ ($\mathbf{m}[\text{fp} + 0]$ and $\mathbf{m}[\text{fp} + 1]$ are used, by convention, to store the caller's pc and fp, to return to the previous context at the end of the function), which perform the following:

1. assert(x < 10)
2. z := x*y + 100
3. assert(z < 1000)

Which can be compiled to:

1. $\mathbf{m}[\text{fp} + 4] = \mathbf{m}[\big[\text{fp} + 2\big]]$ // check that $x$ is "small"
2. $\mathbf{m}[\text{fp} + 2] + \mathbf{m}[\text{fp} + 5] = 9$ // compute $9 - x$
3. $\mathbf{m}[\text{fp} + 6] = \mathbf{m}[\big[\text{fp} + 5\big]]$ // check that $9 - x$ is "small"
4. $\mathbf{m}[\text{fp} + 7] = \mathbf{m}[\text{fp} + 2] * \mathbf{m}[\text{fp} + 3]$ // compute $x.y$
5. $\mathbf{m}[\text{fp} + 8] = \mathbf{m}[\text{fp} + 7] + 100$ // compute $z = x.y + 100$
6. $\mathbf{m}[\text{fp} + 9] = \mathbf{m}[\big[\text{fp} + 8\big]]$ // check that $z$ is "small"
7. $\mathbf{m}[\text{fp} + 8] + \mathbf{m}[\text{fp} + 10] = 999$ // compute $999 - z$
8. $\mathbf{m}[\text{fp} + 11] = \mathbf{m}[\big[\text{fp} + 10\big]]$ // check that $999 - z$ is "small"
9. JUMP with next(pc) = $\mathbf{m}[\text{fp} + 0]$, next(fp) = $\mathbf{m}[\text{fp} + 1]$, condition = 1 // return

#### 2.5.4 Switch statements

Suppose we want a different logic depending on the value $x$ of a given memory cell, where $x$ is known to be $< k$ (if the value $x$ comes from a "hint", don't forget to range-check it).

Each of the $k$ different value leads to a different branch at runtime, represented by a block of code. We want to jump to the correct block of code depending on $x$. One efficient implementation consists in placing our blocks of code at regular intervals, and to jump to a $a + b.x$, where $a$ is the offset of the first block of code (in case $x = 0$), and $b$ is the distance between two consecutive blocks.

Example: During XMSS verification, for each of the $v$ chains, we need to hash a pre-image, a number of times depending on the encoding, but known to be $< w$. Here $k = w$, and the $i - th$ block of code we could jump to will execute $i$ times the hash function (unrolled loop).

# 3 Proving system

## 3.1 Execution table

### 3.1.1 Reduced commitment via logup*

In Cairo each instruction is encoded with 15 boolean flags, and 3 offsets. In the execution trace, this leads to committing to 18 field elements at each instruction.

We can significantly reduce the commitments cost using logup*[7]. In the the execution table, we only need to commit to the pc column, and all the flags / offsets describing the current instruction can be fetched by an indexed lookup argument (for which logup* drastically reduces commitment costs).

#### 3.1.2 Commitment

At each cycle, we commit to 5 (base) field elements:

- pc (program counter)
- fp (frame pointer)
- $\text{addr}_A$, $\text{addr}_B$, $\text{addr}_C$

The following 3 (virtual) columns can be interpreted as indexed lookups, and thus not be committed thanks to logup* (in practice though, the gains are modest compared to the indexed lookup into the bytecode, because memory accesses are less repeated):

- $\text{value}_A = \mathbf{m}[\text{addr}_A]$, $\text{value}_B = \mathbf{m}[\text{addr}_B]$, $\text{value}_C = \mathbf{m}[\text{addr}_C]$

#### 3.1.3 Instruction Encoding

Each instruction is described by 15 field elements:

- 3 operands ($\in \mathbb{F}_p$): $\text{operand}_A$, $\text{operand}_B$, $\text{operand}_C$
- 3 associated flags ($\in \{0, 1\}$): $\text{flag}_A$, $\text{flag}_B$, $\text{flag}_C$
- 8 opcode flags ($\in \{0, 1\}$): ADD, MUL, DEREF, JUMP, POSEIDON_16, POSEIDON_24, DOT_PRODUCT, MULTILINEAR_EVAL
- One multi-purpose operand: AUX

#### 3.1.4 AIR transition constraints

We use transition constraints of degree 5, but it's always possible to make them quadratic with additional columns in the execution table.

We define the following quantities:

- $\nu_A = \text{flag}_A \cdot \text{operand}_A + (1 - \text{flag}_A) \cdot \text{value}_A$
- $\nu_B = \text{flag}_B \cdot \text{operand}_B + (1 - \text{flag}_B) \cdot \text{value}_B$
- $\nu_C = \text{flag}_C \cdot \text{fp} + (1 - \text{flag}_C) \cdot \text{value}_C$

With the associated constraints: $\forall X \in \{A, B, C\} : (1 - \text{flag}_X) \cdot (\text{address}_X - (\text{fp} + \text{operand}_X)) = 0$

---

For addition and multiplication:

- $\text{ADD} \cdot (\nu_B - (\nu_A + \nu_C)) = 0$
- $\text{MUL} \cdot (\nu_B - \nu_A \cdot \nu_C) = 0$

---

When DEREF $= 1$, set $\text{flag}_A = 0$, $\text{flag}_C = 1$ and:

$$\mathbf{m}[\mathbf{m}[\text{fp} + \alpha] + \beta] = \begin{cases} \gamma & \rightarrow \quad \text{AUX} = 1,\ \text{flag}_B = 1 \\ \mathbf{m}[\text{fp} + \gamma] & \rightarrow \quad \text{AUX} = 1,\ \text{flag}_B = 0 \\ \text{fp} & \rightarrow \quad \text{AUX} = 0\ (\text{flag}_B = 1) \end{cases}$$

- $\text{DEREF} \cdot (\text{addr}_C - (\text{value}_A + \text{operand}_C)) = 0$

- $\text{DEREF} \cdot \text{AUX} \cdot (\text{value}_C - \nu_B) = 0$
- $\text{DEREF} \cdot (1 - \text{AUX}) \cdot (\text{value}_C - \text{fp}) = 0$

---

When there is no jump:

- $(1 - \text{JUMP}) \cdot (\text{next(pc)} - (\text{pc} + 1)) = 0$
- $(1 - \text{JUMP}) \cdot (\text{next(fp)} - \text{fp}) = 0$

When JUMP $= 1$, the condition is represented by $\nu_A$:

- $\text{JUMP} \cdot \nu_A \cdot (1 - \nu_A) = 0$
- $\text{JUMP} \cdot \nu_A \cdot (\text{next(pc)} - \nu_C) = 0$
- $\text{JUMP} \cdot \nu_A \cdot (\text{next(fp)} - \nu_B) = 0$
- $\text{JUMP} \cdot (1 - \nu_A) \cdot (\text{next(pc)} - (\text{pc} + 1)) = 0$
- $\text{JUMP} \cdot (1 - \nu_A) \cdot (\text{next(fp)} - \text{fp}) = 0$

Note: the constraint $\text{JUMP} \cdot \nu_A \cdot (1 - \nu_A) = 0$ could be removed, as long as it's correctly enforced in the bytecode.

## 3.2 Precompiles interactions

Each precompile has a dedicated flag in the instruction encoding. For instance, the instruction corresponding the following Poseidon permutation over 16 field elements:

$$\mathit{Poseidon}(\underbrace{\mathbf{m}[24], \ldots, \mathbf{m}[31]}_{\mathbf{m}[3\cdot 8..4\cdot 8]}, \underbrace{\mathbf{m}[80], \ldots, \mathbf{m}[87]}_{\mathbf{m}[10\cdot 8..11\cdot 8]}) = \underbrace{(\mathbf{m}[\mathbf{m}[\text{fp} + 17] \cdot 8], \ldots, \mathbf{m}[\mathbf{m}[\text{fp} + 17] \cdot 8 + 15])}_{\mathbf{m}[\mathbf{m}[\text{fp}+17]\cdot 8..(\mathbf{m}[\text{fp}+17]+2)\cdot 8]}$$

... is encoded by:

- POSEIDON_16 $= 1$
- $\text{flag}_A = 1$, $\text{operand}_A = 3$
- $\text{flag}_B = 1$, $\text{operand}_B = 10$
- $\text{flag}_C = 0$, $\text{operand}_C = 17$

**Bus 1: Execution table → precompiles** The values $(\nu_A, \nu_B, \nu_C) = (3,\ 10,\ \mathbf{m}[\text{fp} + 17])$ must appear in the Poseidon16 table. A grand product argument is used to ensure that all the tuples $(I_{precompile}, \nu_A, \nu_B, \nu_C)$ appearing in the execution table also appear in the corresponding precompile table ($I_{precompile} = 1$ for Poseidon16, 2 for Poseidon24, etc).

**Bus 2: Precompiles → memory** Using the arguments passed by the bus 1, the precompiles can query the memory at the relevant locations, and ensure that the corresponding computation (Poseidon16 in our example) is respected.

Note that when the memory accesses have a power-of-two length, and are aligned by this same power-of-two (as it's the case in Poseidon16, with 8), the lookup precompile → memory is much more efficient (because we can "fold" the memory, in our case we pay the cost of a lookup into a table 8 times smaller then the memory).

# 4 Annex A: batched logup*

We use the same notations as in the logup* paper [7], but instead of a single evaluation point $r$, we now have two of them: $r_1$ and $r_2$, and two corresponding claims: $I^*T(r_1) = e_1$ and $I^*T(r_2) = e_2$.

1. Use a random challenge $\alpha$ to reduce both claims to: $I^*T(r_1) + \alpha \cdot I^*T(r_2) = e_1 + \alpha \cdot e_2$
2. Commit to $I_*(eq_{r_1} + \alpha \cdot eq_{r_2})$
3. Notice that $I^*T(r_1) + \alpha \cdot I^*T(r_2) = \langle I^*T, eq_{r_1} + \alpha \cdot eq_{r_2} \rangle = \langle T, I_*(eq_{r_1} + \alpha \cdot eq_{r_2}) \rangle$
4. Run a sumcheck for $\langle T, I_*(eq_{r_1} + \alpha \cdot eq_{r_2}) \rangle = e_1 + \alpha \cdot e_2$
5. Open $T$ and $I_*(eq_{r_1} + \alpha \cdot eq_{r_2})$

   Finally, ensuring that $I_*(eq_{r_1} + \alpha \cdot eq_{r_2})$ was correctly committed can be easily adapted from the original protocol, by simply using $X \leftarrow eq_{r_1} + \alpha \cdot eq_{r_2}$.

# 5 Annex B: batched WHIR

The current proof system (actually, most proof systems involving lookups) requires to commit data in 2 phases. First, we commit to the memory, and 4 AIR tables (execution, poseidon_16, poseidon_24, dot_produt). Then, after receiving some random challenges, we commit to the "pushforwards" for the two logup* [7] lookups (memory and bytecode). In the end, we open everything at some sampled points. Instead of 2 consecutive and independent WHIR openings, we "batch" them in the following manner:

TODO

# 6 Annex C: simple packing of multilinear polynomials

*Note 1*: It's always possible to reduce $n$ claims about a multilinear polynomial to a single one, using sumcheck. But this trick is not necessary with WHIR, which natively supports an arbitrary number of claims about the committed polynomial. Given the fact that logup* also has a similar capacity (see 4), the current proof system never requires the $n$-to-1 reduction.

*Note 2:* Crucially, the proving cost to add an equality constraint of the form $P((x_1, \dots, x_n)) = y$ to a committed polynomial $P$ via WHIR is $O(2^k)$ where $k = |\{i, x_i \notin \{0, 1\}\}|$ is the number of "non-boolean variables". As a result, adding "sparse" ($(x_i)$ containing boolean values) equality constraints is essentially optimal.

The proving system of a zkVM often involves committing to multiple polynomials, in our case multilinear ones, which must must be opened in the end on potentially multiple points per polynomial.

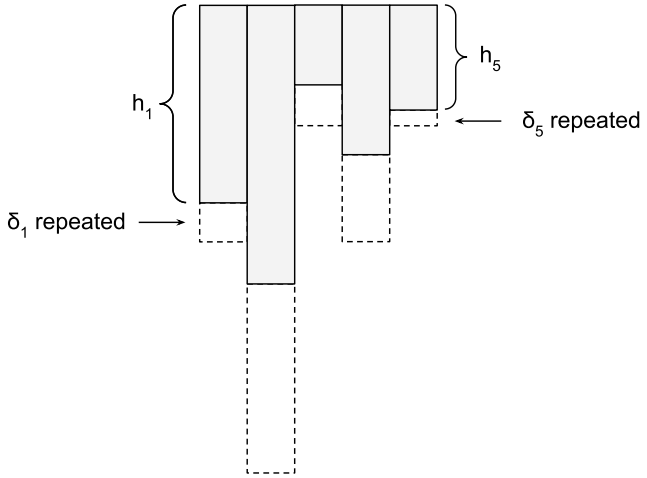
To avoid committing to each polynomial separately (overhead in proof size + prover speed), we need to "pack" multiple polynomials $P_1, \dots, P_n$ in a single one $P$, to which we commit. In the end, we need to reduce evaluations claims on $P_1, \dots, P_n$ to claims on $P$.

In practice, most of the polynomials $P_1, \dots, P_n$ are derived from AIR columns. The height of an AIR column is generally not a power of 2, and the last rows correspond to the repetition of a default value (often zero, but not always, example: poseidon AIR tables). We want to avoid committing as much as possible to this repeated data.

**Summary of the problem:**

- The prover wants to commit to $n$ multilinear polynomials: $P_1, \dots, P_n$
- Prover and verifier both agree on the dimensions: each $P_i$ has $\nu_i$ variables, has height $h_i \leq 2^{\nu_i}$, with default value $\delta_i$, meaning: $\forall x \in \{0, 1\}^{\nu_i}, P_i(x) = \delta_i$ if $x \geq h_i$, interpreted in big-endian.
- Prover will "pack" $P_1, \dots, P_n$ into a single polynomial $P$, which is committed.
- Finally, we must be able to reduce an arbitrary number of evaluation claims on $P_1, \dots, P_n$ into evaluation claims on $P$.

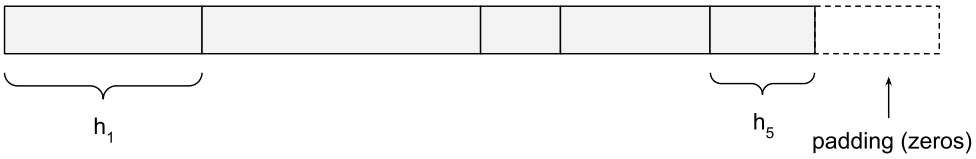
Figure 4: Example with $n = 5$ (the repeated values are represented in dotted line)




## 6.1 "Jagged PCS"

The "jagged PCS" [8] developed by *Succinct* is one possible solution to this problem. The intuitive idea is to pack the non-repeated part of each polynomial together, in a continuous way (no padding at all between two consecutive chunks), and to commit the result as a single multilinear polynomial $P$ (the only padding is at the end of $P$ between the sum of the heights $h = h_1 + \cdots + h_n$ and the next power of 2). Finally, equality constraints on $P_1, \ldots, P_n$ are reduced to a claim on $P$ using a sumcheck (with "selectors" for row / column).

Figure 5: Packing with the "jagged PCS"




## 6.2 Alternative approach without sumcheck (current implem)

We present another way to deal with the packing of multilinears that does not require the sumcheck of the "jagged PCS". The main drawback is a small increase in proof size proportional to $n$ (but, as explained in the next section, we can drastically reduce this overhead, in our case to a few kilobytes).

The condition for this protocol to work is: the verifier must not care if some committed values above the height $h_i$ does not correspond to the default value $\delta_i$. This is the case in practice when committing to AIR tables, or to the memory. It does not mean that the prover can arbitrarily change the values above $h_i$ after commitment, it simply means that the prover is not compelled to commit to $\delta_i$ in this area. (If this condition is not satisfied we can still make this work at a cost of an increase in proof size)

**Idea** (summarized in Figure 6):

- For each polynomial $P_i$:
  - Find the smallest sum of $k$ (think $k = 3$ or 4)[1] powers of two above $h_i$. More precisely, find: $s_i = 2^{\eta_i^1} + \cdots + 2^{\eta_i^k}$ such that $s_i \geq h_i$ and $s_i - h_i$ is small.

[1] We could use a custom number of terms $k_i$ for each polynomial $P_i$, but we use a common one $k$ to simplify the explanation

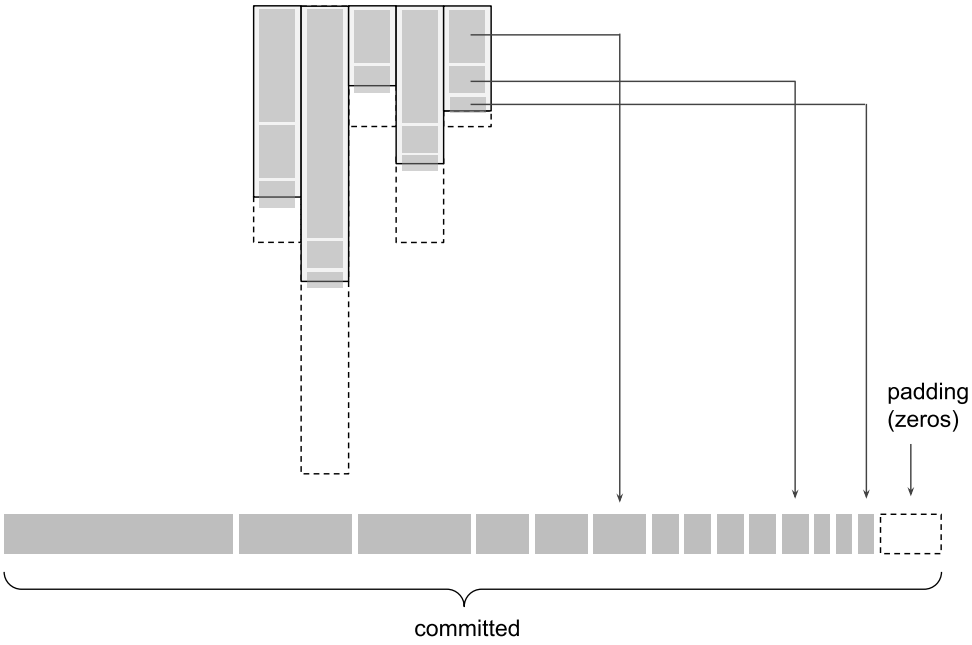
– Cover the initial, non-repeated, part of $P_i$ by $k$ consecutive chunks: $\hat{P}_i^1$ (multilinear in $\eta_i^1$ variable), ..., $\hat{P}_i^k$ (multilinear in $\eta_i^k$ variable), in **decreasing** order ($\eta_i^1 \geq \cdots \geq \eta_i^k$).

- Pack together these $k \cdot n$ chunks, in **decreasing** order, and commit to the resulting multilinear polynomial $P$.
- To open each equality claim $P_i((x_1, \ldots, x_{\nu_i})) = y$, the prover computes the "inner evaluations" and sends them to the verifier: $\hat{P}_i^1(x_{\nu_i - \eta_i^1 + 1}, \ldots, x_{\nu_i})$, ..., $\hat{P}_i^k(x_{\nu_i - \eta_i^k + 1}, \ldots, x_{\nu_i})$. Using those $k$ evaluations (and using the fact that $P_i$ is constant equal to $\delta_i$ on $[s_i..2^{\nu_i}[$), the verifier can "reconstruct" the evaluation of $P_i$ on $(x)$ and ensure equality with the claimed value $y$. Finally, each "inner claim" on $\hat{P}_i^j$ ($1 \leq j \leq k$) is converted into a sparse equality constraint on the committed polynomial $P$ (using boolean variables to "locate" the corresponding chunk).

Optimization: If the verifier already knows the claimed evaluation of a column on a given point: $P_i(x) = y$, we can avoid transmitting one of the "inner evaluations", and typically we skip the first one ($\hat{P}_i^1(x_{\nu_i - \eta_i^1 + 1})$, = the bigger one, which saves some work for the prover.

Example: TODO

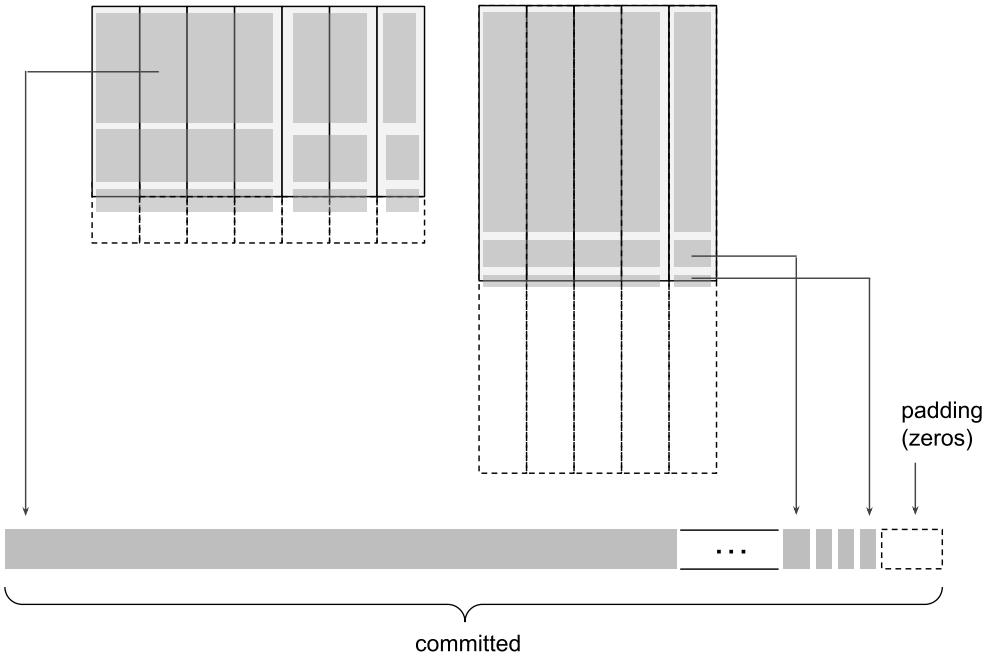
Figure 6: Simple packing, covering the relevant part of each column by power-of-two chunks (in grey)




## 6.3 Reducing the proof size overhead (TODO implem)

Most of the committed columns in the zkVM come from the 2 Poseidon tables, both contain $\approx 400$ columns in total (apart from that, we only commit to $< 20$ columns). Using $k = 3$ above, we get a proof size overhead of $\approx 23KiB$. But we can drastically reduce this using the fact that at the end of the sumcheck-based validity proof for an AIR table, the columns will be queried at a *common point*. As a result, it is possible to commit to our table using a covering of its surface by rectangular chunks, where length and width are both a power of 2 (but potentially different), as described in Figure 7.

Figure 7: Simple packing, covering the AIR tables by power-of-two rectangular chunks (in grey)



# References


[1] J. Drake, D. Khovratovich, M. Kudinov, and B. Wagner, "Hash-based multi-signatures for post-quantum ethereum," Cryptology ePrint Archive, Paper 2025/055, 2025. [Online]. Available: https://eprint.iacr.org/2025/055

[2] D. Khovratovich, M. Kudinov, and B. Wagner, "At the top of the hypercube – better size-time tradeoffs for hash-based signatures," Cryptology ePrint Archive, Paper 2025/889, 2025. [Online]. Available: https://eprint.iacr.org/2025/889

[3] J. Drake, D. Khovratovich, M. Kudinov, and B. Wagner, "Technical note: LeanSig for post-quantum ethereum," Cryptology ePrint Archive, Paper 2025/1332, 2025. [Online]. Available: https://eprint.iacr.org/2025/1332

[4] L. Grassi, D. Khovratovich, and M. Schofnegger, "Poseidon2: A faster version of the poseidon hash function," Cryptology ePrint Archive, Paper 2023/323, 2023. [Online]. Available: https://eprint.iacr.org/2023/323

[5] L. Goldberg, S. Papini, and M. Riabzev, "Cairo – a turing-complete STARK-friendly CPU architecture," Cryptology ePrint Archive, Paper 2021/1063, 2021. [Online]. Available: https://eprint.iacr.org/2021/1063

[6] 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

[7] L. Soukhanov, "Logup*: faster, cheaper logup argument for small-table indexed lookups," Cryptology ePrint Archive, Paper 2025/946, 2025. [Online]. Available: https://eprint.iacr.org/2025/946

[8] T. Hemo, K. Jue, E. Rabinovich, G. Roh, and R. D. Rothblum, "Jagged polynomial commitments (or: How to stack multilinears)," Cryptology ePrint Archive, Paper 2025/917, 2025. [Online]. Available: https://eprint.iacr.org/2025/917