Minimal zkVM for Lean Ethereum (draft 0.6.0)

Contents

1 What is the goal of this zkVM?

Post-quantum signatures are at least an order of magnitude larger than their pre-quantum counterparts, but the migration is necessary to ensure Ethereum's security. Hash-based signatures offer strong security guarantees and conceptual simplicity, making them a promising candidate: leanXMSS at the consensus layer (where statefulness is not an issue, see [1], [2], and [3]), and leanSPHINCS at the execution layer. A promising choice of hash function is Poseidon2 [4], for its snark-friendliness.

Since post-quantum signatures are much larger, we need to aggregate them. However, they lack the algebraic structure that makes aggregation easy for elliptic-curve schemes like BLS. Instead, we can aggregate them using a snark—itself hash-based, keeping everything post-quantum.

Concretely, we need to:

- **Aggregate** hash-based signatures (leanXMSS / leanSPHINCS)
- **Merge** those aggregate signatures → recursive proof

Both tasks mainly require proving a lot of hashes. A minimal zkVM (inspired by Cairo [5]) is useful as glue to handle all the logic: leanVM.

Aggregation and merging are unified in a single program, that attests that a given set of signers have valid signatures for a given message.

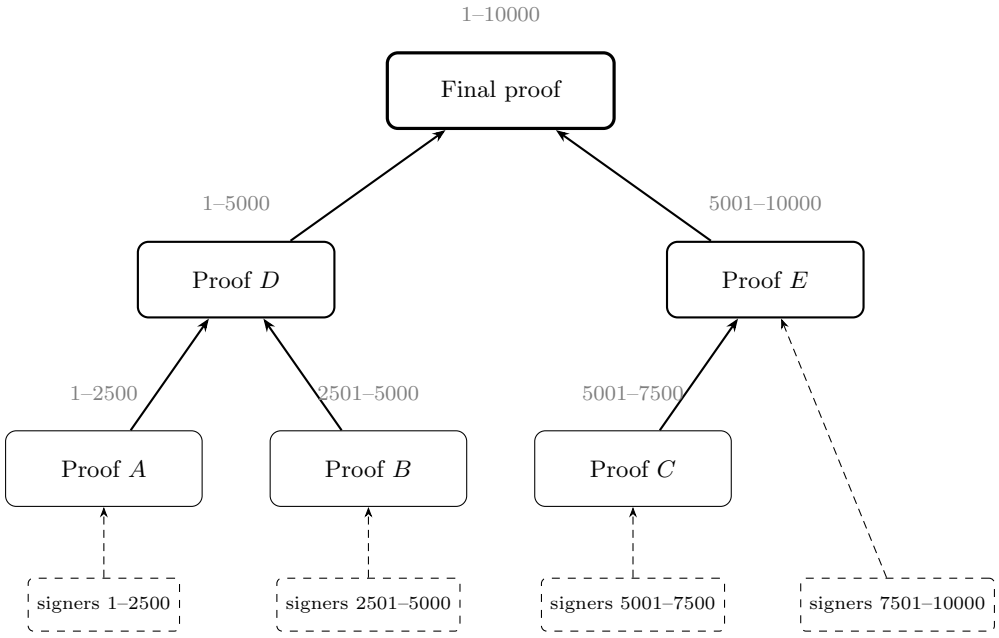


Figure 1: Example of recursive aggregation of 10000 signatures. (Note: overlapping sets of signers are possible.)

2 VM specification

2.1 Field

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

Advantages:

- small field → less Poseidon rounds
- $x \to x^3$ is an automorphism of $\mathbb{F}_p^*$, meaning efficient S-box for Poseidon2 (in BabyBear, it's degree 7)

- $< 2^{31} \rightarrow$ the sum of 2 field elements can be stored in an u32

The small 2-addicity (24) is not a limiting factor in WHIR, thanks to the use of an interleaved Reed Solomon code (example: with an initial folding of 7, and rate 1/2, we can commit up to 2^{30} field elements).

Extension field: $\mathbb{F}_q$, with $q = p^5$: the degree-5 extension enables 128 bits of security in WHIR, with the Johnson bound, thanks to the latest result of [6].

2.2 Memory

- Read-Only Memory
- Word = KoalaBear field element
- Memory size: $M = 2^m$ with $16 \leq m \leq 26$ (m depends on the execution and is communicated at the beginning of the proof).
- The first $M' = 2^{m'}$ memory cells hold the "public input", on which both prover and verifier must agree. This enables to pass the arguments that the leanISA program receives as input (in our case: message to sign and XMSS public keys).

2.3 Registers

- **pc**: program counter : points to the current instruction
- **fp**: frame pointer : points to the start of the current "stack frame" in memory

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

2.4 Instruction Set Architecture

Notations:

- α, β and γ 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

ADD: $\nu_a + \nu_c = \nu_b$ and MUL: $\nu_a \cdot \nu_c = \nu_b$ with:

$$\nu_a = \begin{cases} \alpha \\ \mathbf{m}[\mathbf{fp} + \alpha] \end{cases} \qquad \nu_b = \begin{cases} \beta \\ \mathbf{m}[\mathbf{fp} + \beta] \end{cases} \qquad \nu_c = \begin{cases} \gamma \\ \mathbf{m}[\mathbf{fp} + \gamma] \\ \mathbf{fp} + \gamma \end{cases}$$

2.4.2 DEREF

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

2.4.3 JUMP (Conditional)

Using the following values:

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

... we have the following update rules:

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

2.5 Precompiles

There are two "precompile" instructions: EXTENSION_OP and POSEIDON2, used for special computation (extension field operations and hashing). Each precompile instruction has three (potentially runtime) parameters ν_A, ν_B, ν_C, and a compile-time value `PRECOMPILE_DATA`.

ν_A, ν_B, ν_C are defined similarly to the ADD/MUL instructions, with the additional possibility of $\nu_A = \mathbf{fp} + \alpha$ and $\nu_B = \mathbf{fp} + \beta$, but these two must be used simultaneously (if ν_A is fp-relative, then ν_B must be fp-relative too, and vice versa).

2.5.1 POSEIDON2

Compression of 16 field elements (two blocks of 8) into 8 field elements.

$$\mathbf{m}[\nu_C..\nu_C + 8] = \text{Poseidon2}(\mathbf{m}[\nu_A..\nu_A + 8] | \mathbf{m}[\nu_B..\nu_B + 8]) + \mathbf{m}[\nu_A..\nu_A + 8]$$

`PRECOMPILE_DATA` $= 1$

2.5.2 EXTENSION_OP

EXTENSION_OP enables computations of one these 3 forms in the extension field $\mathbb{F}_q$:

$$\sum_{i=0}^{N-1} (a_i + b_i), \qquad \sum_{i=0}^{N-1} (a_i \cdot b_i), \qquad \prod_{i=0}^{N-1} (a_i b_i + (1 - a_i)(1 - b_i))$$

Details

For $i = 0, \ldots, N-1$, define:

$$b_i = \mathbf{m}[\nu_B + 5i \,..\, \nu_B + 5i + 5] \in \mathbb{F}_q$$

and a_i depends on the mode:

- **BE mode** (`is_be` $= 1$): $a_i = \mathbf{m}[\nu_A + i] \in \mathbb{F}_p$, embedded in $\mathbb{F}_q$ as $[a_i, 0, 0, 0, 0]$.
- **EE mode** (`is_be` $= 0$): $a_i = \mathbf{m}[\nu_A + 5i \,..\, \nu_A + 5i + 5] \in \mathbb{F}_q$.

(Chunks of 5 field elements $\mathbf{m}[x..x+5]$ are interpreted as an extension field element ($\mathbb{F}_q = \mathbb{F}_{p^5}$)).

Three operation modes determine both the per-element computation e_i and the accumulation method:

- **ADD**: $e_i = a_i + b_i$, $\quad \text{result} = \sum_{i=0}^{N-1} e_i$ (additive accumulation)
- **MUL**: $e_i = a_i \cdot b_i$, $\quad \text{result} = \sum_{i=0}^{N-1} e_i$ (additive accumulation = dot product)

- **POLY_EQ**: $e_i = a_i b_i + (1 - a_i)(1 - b_i)$, $\quad$ result $= \prod_{i=0}^{N-1} e_i$ (multiplicative)

The result is stored at $\mathbf{m}[\nu_C \,..\, \nu_C + 5] \in \mathbb{F}_q$.

`PRECOMPILE_DATA` encodes the mode, the operand type, and the length:

$$\texttt{PRECOMPILE\_DATA} = 2 \cdot \texttt{is\_be} + 4 \cdot \texttt{flag\_add} + 8 \cdot \texttt{flag\_mul} + 16 \cdot \texttt{flag\_poly\_eq} + 32 \cdot N$$

where exactly one of `flag_add`, `flag_mul`, `flag_poly_eq` is 1, and $N \geq 1$. This encoding is injective and always disjoint from Poseidon (= 1) and no-precompile (= 0).

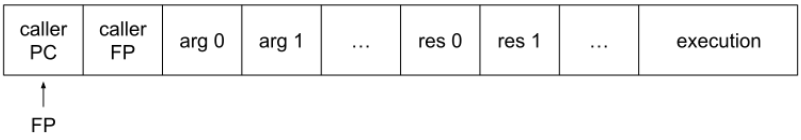
2.6 ISA programming

2.6.1 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 2: Memory layout of a function call



2.6.2 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).

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

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

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

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

Given $t \leq 2^{16} \leq M$, $\mathbf{m}[\mathbf{fp} + x] < M$, $t - 1 - \mathbf{m}[\mathbf{fp} + x] < M$, and $M \leq 2^{26} < p/2$, we have: $\mathbf{m}[\mathbf{fp} + x] < t$.

Note: From the point of view of the prover running the program, some hints are necessary (filling the values of $\mathbf{m}[\mathbf{fp} + i]$ and $\mathbf{m}[\mathbf{fp} + k]$ 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 $x = \mathbf{m}[\mathbf{fp} + 2]$ and $y = \mathbf{m}[\mathbf{fp} + 3]$ ($\mathbf{m}[\mathbf{fp} + 0]$ and $\mathbf{m}[\mathbf{fp} + 1]$ 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. $\mathbf{m}[\mathbf{fp} + 4] = \mathbf{m}[\big[\mathbf{fp} + 2\big]]$ // check that x is "small"
2. $\mathbf{m}[\mathbf{fp} + 2] + \mathbf{m}[\mathbf{fp} + 5] = 9$ // compute $9 - x$
3. $\mathbf{m}[\mathbf{fp} + 6] = \mathbf{m}[\big[\mathbf{fp} + 5\big]]$ // check that $9 - x$ is "small"
4. $\mathbf{m}[\mathbf{fp} + 7] = \mathbf{m}[\mathbf{fp} + 2] * \mathbf{m}[\mathbf{fp} + 3]$ // compute $x.y$
5. $\mathbf{m}[\mathbf{fp} + 8] = \mathbf{m}[\mathbf{fp} + 7] + 100$ // compute $z = x.y + 100$
6. $\mathbf{m}[\mathbf{fp} + 9] = \mathbf{m}[\big[\mathbf{fp} + 8\big]]$ // check that z is "small"
7. $\mathbf{m}[\mathbf{fp} + 8] + \mathbf{m}[\mathbf{fp} + 10] = 999$ // compute $999 - z$
8. $\mathbf{m}[\mathbf{fp} + 11] = \mathbf{m}[\big[\mathbf{fp} + 10\big]]$ // check that $999 - z$ is "small"
9. JUMP with $\text{next}(\mathbf{pc}) = \mathbf{m}[\mathbf{fp} + 0]$, $\text{next}(\mathbf{fp}) = \mathbf{m}[\mathbf{fp} + 1]$, condition = 1 // return

2.6.4 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 Table sizes

Each table has a maximum number of rows (as a power of two):

Table	Max rows
Execution	2^{25}
Extension op	2^{20}
Poseidon	2^{21}

Tables are padded to the next power of two (with a minimum of 2^8 rows). These bounds are critical for soundness: they ensure that logup multiplicities cannot overflow modulo p.

3.2 Arithmetic Intermediate Representation (AIR)

If a table has n columns, a transition constraint is a multivariate polynomial $C(x_1, \ldots, x_n, y_1, \ldots, y_n)$ in $2n$ variables, where $(x_1, \ldots, x_n)$ represents the current row and $(y_1, \ldots, y_n)$ represents the next row. Each table defines a set of such constraints, all of which must evaluate to zero on every pair of consecutive rows. For a table with H rows (indexed $0, \ldots, H-1$), the constraints are evaluated on $(H-1)+1 = H$ pairs:

- $H-1$ consecutive pairs: $(\text{row}_i, \text{row}_{i+1})$ for $i = 0, \ldots, H-2$
- 1 **wrapping pair**: $(\text{row}_{H-1}, \text{row}_{H-1})$ — the last row is paired with itself

The wrapping pair means that on the last row, $\text{next}(x) = x$ for every column x.

TODO: describe how to prove AIR via sumcheck.

3.3 Execution table

3.3.1 Commitment

At each cycle, we commit to 20 (base) field elements:

- **pc** (program counter)
- **fp** (frame pointer)
- addr_A, addr_B, addr_C
- $\text{value}_A = \mathbf{m}[\text{addr}_A]$, $\text{value}_B = \mathbf{m}[\text{addr}_B]$, $\text{value}_C = \mathbf{m}[\text{addr}_C]$
- 12 field elements describing the instruction being executed (see below)

3.3.2 Instruction Encoding

Each instruction is described by 12 field elements ($\mathbb{F}_p$):

- 3 operands: operand_A, operand_B, operand_C
- 5 flags ($\in \{0,1\}$): flag_A, flag_B, flag_C, $\text{flag}_{\mathbf{fp}}^C$, $\text{flag}_{\mathbf{fp}}^{AB}$
- 3 opcode flags: MUL ($\in \{0,1\}$), JUMP ($\in \{0,1\}$), AUX ($\in \{0,1,2\}$) (AUX handles both ADD and DEREF)
- PRECOMPILE_DATA

$\text{flag}_{\mathbf{fp}}^{AB}$ is only used by precompile instructions; for ADD, MUL, DEREF, and JUMP it is always 0.

3.3.3 AIR transition constraints

We use transition constraints of degree 5.

We define the following values:

- $\nu_A = \text{flag}_A \cdot \text{operand}_A + (1 - \text{flag}_A - \text{flag}_{\mathbf{fp}}^{AB}) \cdot \text{value}_A + \text{flag}_{\mathbf{fp}}^{AB} \cdot (\mathbf{fp} + \alpha)$
- $\nu_B = \text{flag}_B \cdot \text{operand}_B + (1 - \text{flag}_B - \text{flag}_{\mathbf{fp}}^{AB}) \cdot \text{value}_B + \text{flag}_{\mathbf{fp}}^{AB} \cdot (\mathbf{fp} + \beta)$
- $\nu_C = \text{flag}_C \cdot \text{operand}_C + (1 - \text{flag}_C - \text{flag}_{\mathbf{fp}}) \cdot \text{value}_C + \text{flag}_{\mathbf{fp}} \cdot (\mathbf{fp} + \gamma)$

With the associated constraints:

- $(1 - \text{flag}_A - \text{flag}_{\mathbf{fp}}^{AB}) \cdot (\text{address}_A - (\mathbf{fp} + \text{operand}_A)) = 0$
- $(1 - \text{flag}_B - \text{flag}_{\mathbf{fp}}^{AB}) \cdot (\text{address}_B - (\mathbf{fp} + \text{operand}_B)) = 0$

- $(1 - \text{flag}_C - \text{flag}_{\mathbf{fp}}^C) \cdot (\text{address}_C - (\mathbf{fp} + \text{operand}_C)) = 0$

Let P_0, P_1, P_2 be the Lagrange basis polynomials at points $0, 1, 2$. Explicitely:

$$P_0(x) = (x-1) \cdot (x-2)/2 \qquad P_1(x) = x \cdot (2-x) \qquad P_2(x) = x \cdot (x-1)/2$$

We define the following values:

- ADD $= P_1(\text{AUX})$
- DEREF $= P_2(\text{AUX})$
- IS_PRECOMPILE $= 1 - (\text{ADD} + \text{MUL} + \text{DEREF} + \text{JUMP})$

IS_PRECOMPILE is used a "bus selector" (see 3.6.2).

PRECOMPILE_DATA encodes the precompile being called, and the parameters of the call:

- No precompile $\rightarrow$ PRECOMPILE_DATA $= 0$
- Poseidon precompile $\rightarrow$ PRECOMPILE_DATA $= 1$
- Extension op precompile $\rightarrow$ PRECOMPILE_DATA $= 2 \cdot \texttt{is\_be} + 4 \cdot \texttt{flag\_add} + 8 \cdot \texttt{flag\_mul} + 16 \cdot \texttt{flag\_poly\_eq} + 32 \cdot N$ (N = length)

For addition, set AUX $= 1$.

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

For multiplication, set MUL $= 1$.

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

For DEREF instructions, set AUX $= 2$, $\text{flag}_A = 0$, $\text{flag}_B = 1$ and flag_C / $\text{flag}_{\mathbf{fp}}^C$ according to whether the third operand is an immediate value, a memory read, or fp-relative.
The relation $\mathbf{m}[\mathbf{m}[\mathbf{fp} + \alpha] + \beta] = \nu_C$ is then captured by:

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

For (conditional) jumps, set JUMP $= 1$, with the following constraints, using $J = \text{JUMP} \cdot \nu_A$:

- $J \cdot (1 - \nu_A) = 0$
- $J \cdot (\text{next}(\mathbf{pc}) - \nu_B) = 0$

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

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

TODO: Verify and (formally) prove soundness.

3.4 Poseidon table

TODO

3.5 Extension op table

See 2.5.2 for the instruction specification.

3.5.1 Trace layout

Each extension field operation occupies N consecutive rows (one row per element pair). Within a group, rows are ordered from first element to last, but the `computation` column accumulates backward: the first row holds the full result, the last row holds only the final element's value. Padding rows have all operation flags set to 0 and `start` $= 1$, `len` $= 1$.

29 base field columns (AIR)													
`is_be`	`start`	$\texttt{f}_{\texttt{add}}$	$\texttt{f}_{\texttt{mul}}$	$\texttt{f}_{\texttt{poly\_eq}}$	`len`	idx_A	idx_B	idx_R	v_A (5 cols)	v_B (5 cols)	`res` (5 cols)	`comp` (5 cols)	`aux`
MUL, BE, $N=4$ ($\nu_A=90$, $\nu_B=211$, $\nu_C=74$): $\texttt{res} = \sum_{i=0}^{3} \boldsymbol{m}[90+i] \cdot \boldsymbol{m}[211+5i..216+5i]$													
1	1	0	1	0	4	90	211	74	$\mathbf{m}[90..95]$	$\mathbf{m}[211..216]$	$\mathbf{m}[74..79]$	$e_0+e_1+e_2+e_3$	138
1	0	0	1	0	3	91	216	74	$\mathbf{m}[91..96]$	$\mathbf{m}[216..221]$	$\mathbf{m}[74..79]$	$e_1+e_2+e_3$	106
1	0	0	1	0	2	92	221	74	$\mathbf{m}[92..97]$	$\mathbf{m}[221..226]$	$\mathbf{m}[74..79]$	e_2+e_3	74
1	0	0	1	0	1	93	226	74	$\mathbf{m}[93..98]$	$\mathbf{m}[226..231]$	$\mathbf{m}[74..79]$	e_3	42
ADD, EE, $N=3$ ($\nu_A=400$, $\nu_B=500$, $\nu_C=300$): $\texttt{res} = \sum_{i=0}^{2} (\boldsymbol{m}[400+5i..405+5i] + \boldsymbol{m}[500+5i..505+5i])$													
0	1	1	0	0	3	400	500	300	$\mathbf{m}[400..405]$	$\mathbf{m}[500..505]$	$\mathbf{m}[300..305]$	$e_0+e_1+e_2$	100
0	0	1	0	0	2	405	505	300	$\mathbf{m}[405..410]$	$\mathbf{m}[505..510]$	$\mathbf{m}[300..305]$	e_1+e_2	68
0	0	1	0	0	1	410	510	300	$\mathbf{m}[410..415]$	$\mathbf{m}[510..515]$	$\mathbf{m}[300..305]$	e_2	36
POLY_EQ, EE, $N=3$ ($\nu_A=600$, $\nu_B=700$, $\nu_C=800$): $\texttt{res} = \prod_{i=0}^{2} [a_i b_i + (1-a_i)(1-b_i)]$													
0	1	0	0	1	3	600	700	800	$\mathbf{m}[600..605]$	$\mathbf{m}[700..705]$	$\mathbf{m}[800..805]$	$e_0 \cdot e_1 \cdot e_2$	112
0	0	0	0	1	2	605	705	800	$\mathbf{m}[605..610]$	$\mathbf{m}[705..710]$	$\mathbf{m}[800..805]$	$e_1 \cdot e_2$	80
0	0	0	0	1	1	610	710	800	$\mathbf{m}[610..615]$	$\mathbf{m}[710..715]$	$\mathbf{m}[800..805]$	e_2	48
Padding row													
0	1	0	0	0	1	0	0	0	$\mathbf{m}[0..5]$	$\mathbf{m}[0..5]$	$\mathbf{m}[0..5]$	0	32

Notes: e_i denotes the per-element result (virtual, not a committed column). v_A stores 5 base-field coordinates $\mathbf{m}[\text{idx}_A..\text{idx}_A+5]$; in BE mode only the first coordinate is used, in EE mode all 5 are used. `aux` is a non-committed bus fingerprinting column, equal to $2 \cdot \texttt{is\_be} + 4 \cdot \texttt{flag\_add} + 8 \cdot \texttt{flag\_mul} + 16 \cdot \texttt{flag\_poly\_eq} + 32 \cdot \texttt{len}$ on each row. The bus selector is the virtual column $\texttt{activation\_flag} = \texttt{start} \cdot \texttt{active}$ (not shown; equals `start` on active rows, 0 on padding).

3.5.2 Columns

29 columns.

All columns are base field ($\mathbb{F}_p$) columns. Extension field values are represented as 5 consecutive base field columns.

- `is_be` $\in \{0, 1\}$: mode flag (1: base × extension, 0: extension × extension)
- `start` $\in \{0, 1\}$: group boundary (1 on first row of each group and on padding, 0 elsewhere)
- `flag_add`, `flag_mul`, `flag_poly_eq` $\in \{0, 1\}$: operation selectors (exactly one is 1 on active rows, all 0 on padding)

- `len`: countdown from N to 1 within each group (1 on padding)
- idx_A, idx_B, idx_R: memory addresses
- v_A (5 cols). In BE mode only the first coordinate is used; in EE mode all 5.
- v_B (5 cols)
- `res` (5 cols) (same on every row of a group)
- `comp` (5 cols): running accumulation; equals `res` on `start` rows

Non-committed columns (used for bus interaction):

- `activation_flag` = `start`·`active`: bus selector (1 on first row of each active group, 0 elsewhere)
- $\texttt{aux} = 2 \cdot \texttt{is\_be} + 4 \cdot \texttt{flag\_add} + 8 \cdot \texttt{flag\_mul} + 16 \cdot \texttt{flag\_poly\_eq} + 32 \cdot \texttt{len}$

3.5.3 Memory lookups

- $v_A = \mathbf{m}[\text{idx}_A .. + 5]$
- $v_B = \mathbf{m}[\text{idx}_B .. + 5]$
- $\texttt{res} = \mathbf{m}[\text{idx}_R .. + 5]$

3.5.4 Bus interaction

On rows where `activation_flag` = 1 (i.e. `start` = 1 and `active` = 1), the table PULLs $(\texttt{aux}, \text{idx}_A, \text{idx}_B, \text{idx}_R)$ from the precompile bus. The execution table PUSHes a matching tuple for each EXTENSION_OP instruction.

The `aux` encoding ensures both the mode and the length are bound to the bus data. Since `is_be`, `flag_add`, `flag_mul`, and `flag_poly_eq` are constrained to be boolean, and since the length is constrained to be $\leq 2^{20}$ (by Lemma 1), no overflow can occur modulo p and the `aux` value is unique for each combination of parameters, which enforces that all values of PRECOMPILE_DATA sent by the execution table are correctly received by the extension_op table.

3.5.5 AIR constraints

Degree 6.

Value selection: In BE mode, v_A is treated as a base field element (only the first of its 5 coordinates matters); in EE mode all 5 coordinates are used. Formally, for $k = 1, \ldots, 4$: $v_A[k]$ is multiplied by $(1 - \texttt{is\_be})$ in the constraints, so only $v_A[0]$ survives in BE mode.

Let `active` = `flag_add` + `flag_mul` + `flag_poly_eq` (1 on active rows, 0 on padding), and define the per-element results (virtual):

$$
\begin{aligned}
e_{\text{add}} &= v_A + v_B \\
e_{\text{mul}} &= v_A \cdot v_B \\
e_{\text{peq}} &= v_A \cdot v_B + (1 - v_A)(1 - v_B)
\end{aligned}
$$

Let $\texttt{comp\_tail} = \text{next}(\texttt{comp}) \cdot (1 - \text{next}(\texttt{start}))$ (zero when the next row is a new group).

Boolean constraints:

1. $\texttt{is\_be} \cdot (1 - \texttt{is\_be}) = 0$
2. $\texttt{start} \cdot (1 - \texttt{start}) = 0$
3. $\texttt{flag\_add} \cdot (1 - \texttt{flag\_add}) = 0$
4. $\texttt{flag\_mul} \cdot (1 - \texttt{flag\_mul}) = 0$
5. $\texttt{flag\_poly\_eq} \cdot (1 - \texttt{flag\_poly\_eq}) = 0$

Accumulation constraint:

6. $\texttt{f}_{\texttt{add}} \cdot (\texttt{comp} - e_{\text{add}} - \texttt{comp\_tail}) = 0$ (additive ADD)
7. $\texttt{f}_{\texttt{mul}} \cdot (\texttt{comp} - e_{\text{mul}} - \texttt{comp\_tail}) = 0$ (additive MUL)
8. $\texttt{f}_{\texttt{peq}} \cdot (\texttt{comp} - e_{\text{peq}} \cdot (\texttt{comp\_tail} + \text{next}(\texttt{start}))) = 0$ (multiplicative POLY_EQ)

Result constraint:

9. $\texttt{start} \cdot (\texttt{comp} - \texttt{res}) = 0$ (on start rows, computation equals result)

Intra-group consistency:

10. $(1 - \text{next}(\texttt{start})) \cdot (\texttt{len} - \text{next}(\texttt{len}) - 1) = 0$ (len countdown)
11. $(1 - \text{next}(\texttt{start})) \cdot (\texttt{is\_be} - \text{next}(\texttt{is\_be})) = 0$ (mode consistent)
12. $(1 - \text{next}(\texttt{start})) \cdot (\texttt{f}_{\texttt{add}} - \text{next}(\texttt{f}_{\texttt{add}})) = 0$
13. $(1 - \text{next}(\texttt{start})) \cdot (\texttt{f}_{\texttt{mul}} - \text{next}(\texttt{f}_{\texttt{mul}})) = 0$
14. $(1 - \text{next}(\texttt{start})) \cdot (\texttt{f}_{\texttt{peq}} - \text{next}(\texttt{f}_{\texttt{peq}})) = 0$
15. $(1 - \text{next}(\texttt{start})) \cdot (\text{next}(\text{idx}_A) - \text{idx}_A - (\texttt{is\_be} + (1 - \texttt{is\_be}) \cdot 5)) = 0$
16. $(1 - \text{next}(\texttt{start})) \cdot (\text{next}(\text{idx}_B) - \text{idx}_B - 5) = 0$

Boundary:

17. $\text{next}(\texttt{start}) \cdot (\texttt{len} - 1) = 0$ (when next row is a group start, current len must be 1)

Lemma 1. On every row r of the dot product table, $\texttt{len}_r \leq H - r$ (as an integer in $\{0, \ldots, p-1\}$), where $H \leq 2^{21}$ is the table height. In particular, $\texttt{len}_r \leq H$.

Proof. By backward induction on r, from $r = H - 1$ down to $r = 0$.

Base case ($r = H - 1$): On the wrapping pair (row $H-1$ paired with itself, see Section 3.2), $\text{next}(x) = x$ for every column x. Constraint 10 becomes $(1-\texttt{start})\cdot(\texttt{len}-\texttt{len}-1) = -(1-\texttt{start}) = 0$, forcing $\texttt{start}_{H-1} = 1$. Then constraint 17 becomes $1\cdot(\texttt{len}-1) = 0$, giving $\texttt{len}_{H-1} = 1 = H-(H-1)$.

Inductive step ($r < H - 1$): Assume $\texttt{len}_s \leq H - s$ for all $s > r$. The pair (row r, row $r+1$) gives two cases depending on $\texttt{start}_{r+1} \in \{0, 1\}$ (constraint 2):

- If $\texttt{start}_{r+1} = 1$: constraint 17 gives $1 \cdot (\texttt{len}_r - 1) = 0$, so $\texttt{len}_r = 1 \leq H - r$.
- If $\texttt{start}_{r+1} = 0$: constraint 10 gives $(1 - 0) \cdot (\texttt{len}_r - \texttt{len}_{r+1} - 1) = 0$, so $\texttt{len}_r = \texttt{len}_{r+1} + 1$ in $\mathbb{F}_p$. By induction, $\texttt{len}_{r+1} \leq H - (r + 1)$, so $\texttt{len}_{r+1} + 1 \leq H - r$. Since $H - r \leq H \leq 2^{21} < p$, the addition does not overflow modulo p, so $\texttt{len}_r = \texttt{len}_{r+1} + 1 \leq H - r$ as integers.

□

TODO: Verify and (formally) prove soundness.

3.6 Data flow between tables / memory

Lemma 2. Let $a_0, a_1, \ldots, a_{n-1}$ be pairwise distinct poles in $\mathbb{F}_q$, and let $m_0, m_1, \ldots, m_{n-1}$ be an associated list of multiplicities in $\{0, 1, \ldots, p-1\}$. Consider the rational function:

$$P(X) = \sum_{i=0}^{n-1} \frac{m_i}{X - a_i}$$

Except with probability n/q, if $P(\alpha) = 0$ for a random $\alpha \in \mathbb{F}_q$, then all multiplicities $m_i = 0$.

3.6.1 Indexed Lookup into Memory

We use logup [7], in its indexed form, to allow tables to perform lookups into the read-only memory.

Let $\mathcal{T}$ denote the set of all tables in the system. For each table $T \in \mathcal{T}$ with H_T rows, let n_T denote the number of memory lookups. Each lookup $i < n_T$ consists of an **index column** $\text{col}_{\text{index},T,i}$ and a **value column** $\text{col}_{\text{val},T,i}$.

The rule to enforce is the following:

$$\forall T \in \mathcal{T}, \forall i < n_T, \forall j < H_T : \quad \text{col}_{\text{val},T,i}(j) = \mathbf{m}[\text{col}_{\text{index},T,i}(j)]$$

Implicitly, we must also have $\text{col}_{\text{index},T,i}(j) < M$ (the memory size).

The prover initially commits to a multilinear polynomial acc, having the same size as the memory, such that (in the honest case) for every $k < M$:

$$acc[k] = \sum_{T \in \mathcal{T}} \sum_{i < n_T} |\{j < H_T : \text{col}_{\text{index},T,i}(j) = k\}|$$

i.e., $acc[k]$ represents the total number of times address k is accessed by the lookups across all tables.

The verifier sends a random challenge $\alpha \in \mathbb{F}_q$ (TODO quantify soundness error). Let $N = \sum_{T \in \mathcal{T}} n_T \cdot H_T$ be the total number of memory lookups. Assuming $N < p$ (to avoid overflow), the indexed lookup into memory is valid, except with probability $(N + M)/q$, if for a randomly sampled $X \in \mathbb{F}_q$:

$$\sum_{T \in \mathcal{T}} \sum_{i < n_T} \sum_{j < H_T} \frac{1}{X - (\text{col}_{\text{index},T,i}(j) + \alpha \cdot \text{col}_{\text{val},T,i}(j))} = \sum_{k < M} \frac{acc(k)}{X - (k + \alpha \cdot \mathbf{m}[k])}$$

This can be computed via GKR, as introduced in [8].

3.6.2 Precompile bus: Data flow between tables

See OpenVM [9], definition 2.2.2, for more details.

At each cycle when IS_PRECOMPILE = 1, the execution table PUSHes

$$\sigma = (\text{PRECOMPILE\_DATA}, \nu_A, \nu_B, \nu_C)$$

to the "precompile bus".

In case of Poseidon call, PRECOMPILE_DATA = 1 and ν_A, ν_B, ν_C represent the left input address, the right input address, and the output address (3 pointers to a chunk of 8 field elements in memory).

In case of extension op call, PRECOMPILE_DATA $= 2 \cdot \texttt{is\_be} + 4 \cdot \texttt{flag\_add} + 8 \cdot \texttt{flag\_mul} + 16 \cdot \texttt{flag\_poly\_eq} + 32 \cdot N$ encodes the mode, the operation, and the length N, and ν_A, ν_B, ν_C represent the starting address of the first operand, the starting address of the second operand, and the address of the result.

Each such tuple σ (counting multiplicity) must be PULLED by one of the precompile tables. Given that the Poseidon table only pulls tuples with PRECOMPILE_DATA = 1, and the extension op table only pulls tuples with PRECOMPILE_DATA ≥ 32, the bus is effectively partitioned into two disjoint sub-buses, one for each precompile.

Similarly to the IS_PRECOMPILE (virtual) column for the execution table, each one of the two precompile tables has a boolean (virtual) column acting as a bus selector, indicating at which rows the table is pulling from the bus (1) or not (0).

Balance rule: At the end of execution, for each tuple σ, the number of pushes must equal the number of pulls accross the 3 tables: execution, poseidon, and extension op.

Proving system (logup):

Let $\mathcal{T}$ denote the set of all tables in the system. Each table $T \in \mathcal{T}$ has H_T rows. The bus interaction of the table T is defined by:

- $\text{dir}_T \in \{+1, -1\}$: the direction ($+1$ for PUSH, -1 for PULL)
- $\text{sel}_T : [0, H_T) \to \{0, 1\}$: the selector (virtual) column
- $\sigma_T = (\sigma_0, \ldots, \sigma_{k-1})$: the k data columns

The verifier sends a random challenge $\alpha \in \mathbb{F}_q$. Define the encoding of a tuple $\sigma = (\sigma_0, \ldots, \sigma_{k-1})$ by:

$$\text{encode}(\sigma_0, \ldots, \sigma_{k-1}) = \sum_{i=0}^{k-1} \alpha^i \cdot \sigma_i$$

Except with negligible probability probability (TODO quantify precisely), the bus is balanced if for a randomly sampled $X \in \mathbb{F}_q$:

$$\sum_{T \in \mathcal{T}} \sum_{j=0}^{H_T - 1} \frac{\text{dir}_T \cdot \text{sel}_T(j)}{X - \text{encode}(\sigma_T(j))} = 0$$

Overflow constraint: We must ensure that multiplicities do not overflow modulo p. This is enforced by limitting the maximum height of each table.

As with the indexed lookup into memory, this sum can be computed efficiently via GKR [8]. In practice, all logup instances (bus balance and memory lookups) are batched into a single GKR instance, using an addition field element passed to the encoding tuple for domain separation.

TODO: Unified vision in which memory is just another table, and memory accesses are just bus interactions?

A detailled soundness analysis can be found in Soundness of Interactions via LogUp.

3.7 Simple stacking 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.

Note 2: Crucially, the proving cost to add an equality constraint of the form $P((x_1, \ldots, x_n)) = y$ 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.

In order to commit to multiple univariate polynomials with FRI, each polynomial must be FFT-ed + Merkle-committed. Even if it's possible to have some batching at the Merkle tree level (see 'MMCS' in Plonky3), the proof size for multiple, complex AIR tables quickly reach the megabyte scale.

With a multilinear PCS (such as WHIR), we can "concatenate" multiple multilinear polynomials into a single one, and commit to it once (offering significant proof size savings).

More details: Given n multilinear polynomials $P_1, \ldots, P_n$ with $\nu_1, \ldots, \nu_n$ variables respectively, we order them from the **largest to the smallest** and concatenate their respective evaluation (on the boolean hypercube):

$$P = [P_1(\{0, 1\}^{\nu_1}) \| P_2(\{0, 1\}^{\nu_2}) \| \ldots \| P_n(\{0, 1\}^{\nu_n})]$$

After padding with zeros to the next power of two, we can interpret the result as the evaluations (on the boolean hypercube) of a multilinear polynomial, with $\nu = \lceil \log_2(\sum_i 2^{\nu_i}) \rceil$ variables, and commit to it.

To reduce an evaluation claim on an "inner" (smaller) polynomial P_i to a claim on the "outer" (larger) polynomial P, we use **boolean selectors**.

Example: Consider 3 multilinear polynomials P_1, P_2, P_3 with 4, 3, and 2 variables respectively. The concatenated polynomial P, that we commit, has $5 = \lceil \log_2(2^4 + 2^3 + 2^2) \rceil$ variables.

- $P_1(x_1, x_2, x_3, x_4) = P(0, x_1, x_2, x_3, x_4)$
- $P_2(x_1, x_2, x_3) = P(1, 0, x_1, x_2, x_3)$
- $P_3(x_1, x_2) = P(1, 1, 0, x_1, x_2)$

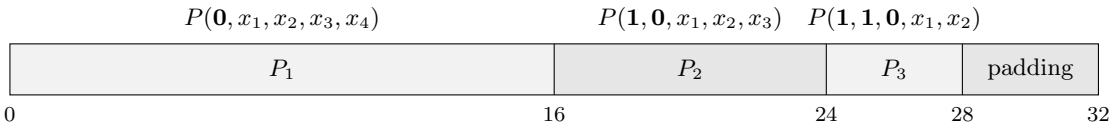


Figure 3: Simple stacking of P_1, P_2, P_3 into a single polynomial P

Advantage of this approach: simplicity.
Drawback: padding overhead, i.e. it does not take advantage of the potential repetions at the end of each inner polynomial's evaluations.

There are alterntive ways to handle the stacking of multiple multilinear polynomials:

- **Jagged PCS** [10]: No padding overhead, at the cost of an additional sumcheck.
- **Per-polynomial chunking**: decompose the non-repeated par of each inner polynomial into a small (think 3 or 4) number of power-of-two sized chunks, and concatenate all these chunks into a single large polynomial.

Note 3: A meticulous implementation of WHIR could also take advantage of any potential repetitions in the committed polynomial, both at the FFT and the sumcheck level, leaving the Merklelisation as the main overhead.

4 Recursive Aggregation

The recursive aggregation program (see Figure 1) attests that a set of public keys have valid signatures for a given message. It partitions the signers into $n_{\text{rec}} + 1$ sources: one **direct source** whose XMSS signatures are verified explicitly, and $n_{\text{rec}} \geq 0$ **recursive sources**, each accompanied by a sub-proof that is recursively verified.

For each recursive source, the program runs the leanVM verifier on the provided sub-proof, confirming that the sub-proof itself is a valid recursive aggregation proof covering the source's signers. When $n_{\text{rec}} = 0$, the program simply verifies all signatures directly.

4.1 Bytecode evaluation claims

A key subtlety arises from recursion. Every leanVM proof is tied to a specific *bytecode* — the program that was executed, represented as a multilinear polynomial. In our case, there is a single program (that both contain signature verification and recursion algorithms). The leanVM verification algorithm requires evaluating the bytecode polynomial at a random point.

When the recursive aggregation program verifies a sub-proof, it runs the leanVM verifier *inside* the program. This verification produces a bytecode evaluation claim. For efficiency, rather than checking it inside the program (with a PCS), the claim is forwarded to the public input, to be checked externally.

However, each sub-proof is potentially itself a recursive aggregation, and may have already forwarded its own bytecode claim in the same way. So for each of the n_{rec} sub-proofs, **two** bytecode claims arise:

1. **Inner-public-memory claim**: The bytecode claim that the sub-proof forwarded from its own recursive verifications (read from the sub-proof's public input).

2. **Inner-proof claim**: The new bytecode claim produced by verifying the sub-proof itself.

After verifying all n_{rec} sub-proofs, a fresh Fiat–Shamir transcript is initialized and fed with all $2n_{\text{rec}}$ evaluation points and claimed values. This transcript produces a random linear combination challenge, and the $2n_{\text{rec}}$ claims are then batched into a single one via sumcheck, yielding a reduced claim. This reduced claim is written to the public input for the next level of recursion.

When $n_{\text{rec}} = 0$ (no recursion), the bytecode claim in the public input is set to zero.

4.2 Signer partitioning

In the following, we assume the message being signed is common to all signers (typically the case at Ethereum's consensus layer), but the scheme can be naturally adapted to distinct messages.

Consider an aggregation of n_{sigs} (distinct) signatures. The n_{sigs} (distinct) public keys are part of the "public input" in memory. The aggregation program must ensure that each public key is verified by at least one of the $1 + n_{\text{rec}}$ sources (duplications allowed: some public keys may be verified by multiple sources).

Let n_{dup} be the total number of duplicated public keys across sources, counted with multiplicity. To ensure ensure valid partionning, we use the following algorithm:

- Initialize a counter $c \leftarrow 0$.
- Initialize a (write-once) array B of size $n_{\text{total}} = n_{\text{sigs}} + n_{\text{dup}}$.
- For each source s, for each signature index $i \in [0, n_{\text{total}})$ verified by s, write $B[i] \leftarrow c$ and increment c.
- At the end, assert $c = n_{\text{total}}$.

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] E. Ben-Sasson, D. Carmon, U. Haböck, S. Kopparty, and S. Saraf, "On proximity gaps for reed–solomon codes," Cryptology ePrint Archive, Paper 2025/2055, 2025. [Online]. Available: https://eprint.iacr.org/2025/2055

[7] U. Haböck, "Multivariate lookups based on logarithmic derivatives," Cryptology ePrint Archive, Paper 2022/1530, 2022. [Online]. Available: https://eprint.iacr.org/2022/1530

[8] S. Papini and U. Haböck, "Improving logarithmic derivative lookups using GKR," Cryptology ePrint Archive, Paper 2023/1284, 2023. [Online]. Available: https://eprint.iacr.org/2023/1284

[9] O. CONTRIBUTORS, "Openvm whitepaper," 2025. [Online]. Available: https://openvm.dev/whitepaper.pdf

[10] 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