Minimal zkVM for Lean Ethereum (draft 0.6.0)

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 1 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 ← (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

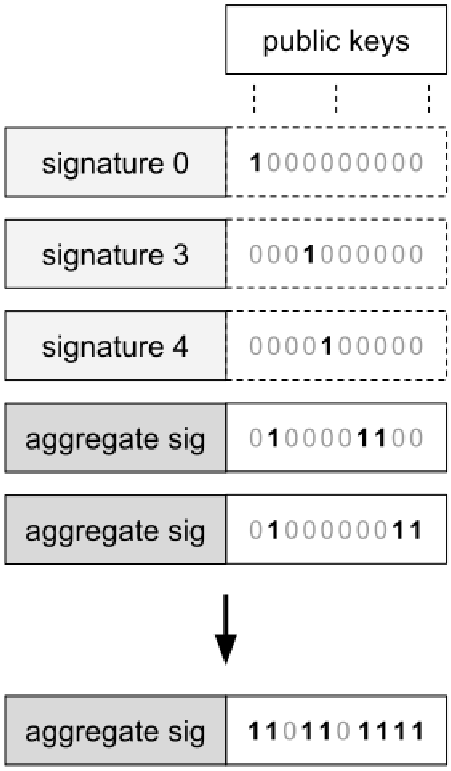
2.1 Field

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

Advantages:

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

Figure 1: AggregateMerge visualized.



- $< 2^{31}$ → 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.

Extension field: of degree 5 : 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 $16 \leq m \leq 29$ (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
- fp: frame pointer : points to the start of the current stack

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

$a + c = b$ 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 2 Precompiles

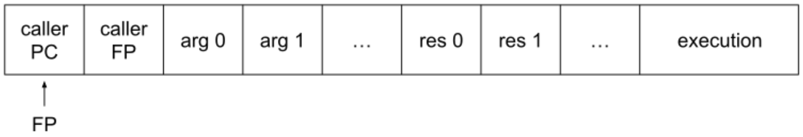
1. **POSEIDON**: Poseidon2 permutation over 16 field elements.
2. **DOT_PRODUCT**: Computes a dot product between either:
   - two slices of extension field elements
   - one slice of base field elements, and one slice of extension field elements

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 2: 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. $\mathbf{m}[\mathbf{m}[\text{fp} + x]] + \mathbf{m}[\text{fp} + j] =$ (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 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)

3.1.2 Instruction Encoding

Each instruction is described by 12 field elements:

- 3 operands ($\in \mathbb{F}_p$): operand_A, operand_B, operand_C
- 3 associated flags ($\in \{0, 1\}$): flag_A, flag_B, flag_C
- 5 opcode flags ($\in \{0, 1\}$): ADD, MUL, DEREF, JUMP, PRECOMPILE_INDEX
- 1 multi-purpose operand: AUX

3.1.3 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 $\text{DEREF} = 1$, 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 $\text{JUMP} = 1$, the condition is represented by ν_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 Data flow between tables / memory

Lemma 1. 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 $m_i = 0$.

3.2.1 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].

3.2.2 Buses: Data flow between tables

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

To each table is associated a list of buses. Declaring a bus for a given table requires specifying:

1. **Direction**: either Pull or Push
2. **Bus index**: can be either a constant or given by a column
3. **Data columns**: a list (of arbitrary size) of columns defining which data is passed to the bus
4. **Selector column**: enforced via AIR constraint to only contain Boolean values ($\in \{0, 1\}$), indicating at which row the bus is active (1) or silenced (0)

Balance rule: At the end of execution, every bus (identified by its index) must be *balanced* across all tables: for each data tuple, the number of pushes must equal the number of pulls.

Proving system (logup):

Let $\mathcal{T}$ denote the set of all tables in the system. For each table $T \in \mathcal{T}$ with H_T rows, let $\mathcal{B}_T$ denote the set of buses associated with T. For each bus $b \in \mathcal{B}_T$, we have:

- $\text{dir}_{T,b} \in \{+1, -1\}$: the direction ($+1$ for Push, -1 for Pull)
- $\text{sel}_{T,b} : [0, H_T) \to \{0, 1\}$: the selector column
- $\text{bus\_idx}_{T,b} : [0, H_T) \to \mathbb{F}_p$: the bus index (assumed to be a column here, but could also be a constant)
- $\text{data}_{T,b} = (d_0, d_1, \ldots, d_{k-1})$: the k data columns

The verifier sends a random challenge $\alpha \in \mathbb{F}_q$ (TODO quantify soundness error). For a bus with index β and data tuple $(c_0, c_1, \ldots, c_{k-1})$, define the encoding:

$$\text{encode}(\beta, c_0, \ldots, c_{k-1}) = \beta + \sum_{i=0}^{k-1} \alpha^{i+1} \cdot c_i$$

Except with probability N/q, where $N = \sum_{T\in\mathcal{T}} |\mathcal{B}_T| \cdot H_T$ is the total number of bus interactions, the bus balance constraint is satisfied if for a randomly sampled $X \in \mathbb{F}_q$:

$$\sum_{T\in\mathcal{T}} \sum_{b\in\mathcal{B}_T} \sum_{j=0}^{H_T-1} \frac{\mathrm{dir}_{T,b} \cdot \mathrm{sel}_{T,b}(j)}{X - \mathrm{encode}\big(\mathrm{bus\_idx}_{T,b}(j), \mathrm{data}_{T,b}(j)\big)} = 0$$

Overflow constraint: For soundness, we must ensure that multiplicities do not overflow modulo p. This requires that for each bus index β, the total number of interactions (pushes and pulls combined) is strictly less than p:

$$\forall\beta : \quad \sum_{\substack{T\in\mathcal{T}, b\in\mathcal{B}_T \\ \mathrm{bus\_idx}_{T,b}=\beta}} H_T < p$$

As with the indexed lookup into memory, this sum can be computed efficiently via GKR [8]. In practice, all logup sums (bus balance and memory lookups) are batched into a single GKR instance, using random linear combination.

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.

4 Annex: 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 $P((x_1, \dots, x_n)) = y$ 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.

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 $P_1, \dots, P_n$ with $\nu_1, \dots, \nu_n$ variables respectively, we order them from the **largest to the smallest** and concatenate their respective evaluation (on the boolean hypercube):

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

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.

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

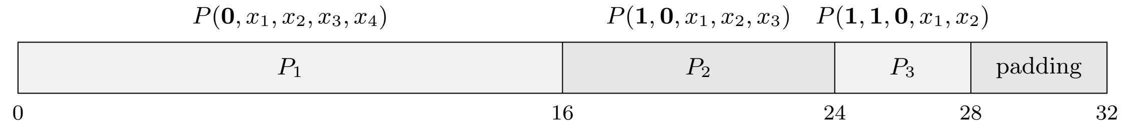

$P(\mathbf{0}, x_1, x_2, x_3, x_4)$
$P(\mathbf{1}, \mathbf{0}, x_1, x_2, x_3)$
$P(\mathbf{1}, \mathbf{1}, \mathbf{0}, x_1, x_2)$
P_1
P_2
P_3
padding
0
16
24
28
32

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.

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