Minimal zkVM for Lean Ethereum (draft 0.3.1)

The views expressed herein are exclusively those of Tom Wambsgans, and do not engage any other party.

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[1]
- 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
 1: Check: bitfield = $\bigcup_{i=0}^{s-1}$ sub_bitfields[i]
 2:                  ▷ Verify the first $s-1$ sub_bitfields using aggregate_proofs:
 3: for $i \leftarrow 0$ to $s-2$ do
 4:     inner_public_input ← (pub_keys, sub_bitfields[i], msg)
 5:     snark_verify("AggregateMerge", inner_public_input, aggregate_proofs[i])
 6: end for
 7:                           ▷ Verify the last sub_bitfields using signatures
 8: $k \leftarrow 0$
 9: for $i \leftarrow 0$ to $n-1$ do
10:     if sub_bitfields[s-1][i] = 1 then
11:         signature_verify(msg, pub_keys[i], signatures[k])
12:         $k \leftarrow k+1$
13:     end if
14: end for
```

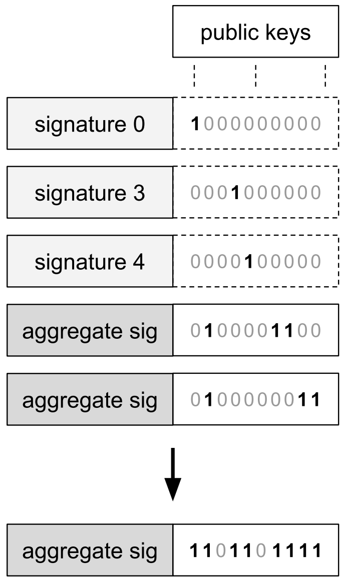
2 Specification

2.1 Field

2.1.1 Base field

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

Figure 1: AggregateMerge visualized.



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)
- $< 2^{31}$ → 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

$\boxed{\text{Extension of dimension 5 or 6}}$

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

2.2 Memory

$\boxed{\text{Read-Only Memory}}$

- Advantage = easier + cheaper to prove
- Drawback = Less convenient to write a program (note that we only have one program to write: AggregateMerge)

Directly using a read-write memory in the proof system would probably require Offline Memory Checking (Spice [7]) or a memory argument based on reordering, which in both case require a range check at each memory operation.

2.3 Registers

- $\boxed{\text{pc: program counter}}$
- $\boxed{\text{fp: frame pointer}}$: points to the start of the current stack

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

2.4 Instruction Set Architecture

α, β and γ represent parameters of the instructions (i.e. immediate value operands)

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 Jump)

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

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

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: a precompile to compute the dot product between two slices of extension field elements.
4. MULTILINEAR_EVAL: a precompile where a chunk of memory is interpreted as a multilinear polynomial with coefficients in the base field, and is evaluated at a point over the extension field. $\boxed{\text{The implementation complexity of this precompile is essentially free:}}$ We can simply add "sparse" equality constraints with WHIR. No AIR table needed.

> The DOT_PRODUCT precompile is used for recursion:
>
> - 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.
>
> The MULTILINEAR_EVAL precompile is also used for recursion:

- In WHIR, evaluating the merkle leaves at the initial round: ≈ 100 multilinear evaluations in ≈ 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)

We use the notation $\mathbf{m_8}[a] = \mathbf{m}[8 \cdot a \dots 8 \cdot (a+1)]$ (vectorized memory by chuncks of 8), and $\|$ for concatenation.

The precompiles are defined by:

- POSEIDON_16$(\mathbf{m_8}[a] \parallel \mathbf{m_8}[b]) = \mathbf{m_8}[c] \parallel \mathbf{m_8}[c+1]$
- POSEIDON_24$(\mathbf{m_8}[a] \parallel \mathbf{m_8}[a+1] \parallel \mathbf{m_8}[b]) = \cdots \parallel \mathbf{m_8}[c]$
- DOT_PRODUCT$((\mathbf{m_8}[a], \dots, \mathbf{m_8}[a+n]), (\mathbf{m_8}[b], \dots, \mathbf{m_8}[b+n]) = \mathbf{m_8}[c]$
- MULTILINEAR_EVAL$((m[a \cdot 2^n], \dots, m[a \cdot 2^n + 2^n - 1]), (\mathbf{m_8}[b], \dots, \mathbf{m_8}[b+n])) = \mathbf{m_8}[c]$

With:

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

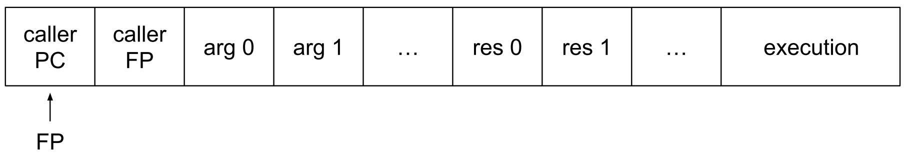
Note: For the two last precompile, for $x = a, b$ or c, only the first D base field elements contained in $\mathbf{m_8}[x]$ are considered, where D is the dimension of the extension field. TODO: avoid sparse representation when $D < 8$.

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 to 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



Optimization idea: if we call two times the same function with exactly the same arguments, we can reuse the same memory frame, instead of a new allocation.

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

To check that a given memory cell x is $< 2^k$:

1. Receive as "hint" the decomposition in bits: $b_0, b_1, \dots, b_{k-1}$
2. For each of the supposed bit b, check $b \cdot (1 - b) = 0$
3. Compute: $2^0 \cdot b_0$, $2^1 \cdot b_1$, $\dots$, $2^{k-1} \cdot b_{k-1}$
4. Sum those values, and check that the result equals x

Cost: $4k$ memory cells and $4k$ cycles.

Range checks that are not a power of two can also be handled via bit-decomposition (but more expensive).

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).

2.6 AIR for the ISA

2.6.1 Logup* to reduce commitment costs

In Cairo each instruction is encoded in one (optionally two) field element, in which 15 boolean flags, and 3 offsets, are packed. In the execution trace, this leads to committing to 18 field elements at each instruction (unpacking flags and offsets).

We can significantly reduce the commitments cost using logup*[8]. 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).

2.6.2 Instruction Encoding

Each instruction is described by 15 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
- 8 opcode flags ($\in \{0, 1\}$): ADD, MUL, DEREF, JUMP, POSEIDON_16, POSEIDON_24, DOT_PRODUCT, MULTILINEAR_EVAL
- One multi-purpose operand: AUX

2.6.3 Execution table

$\boxed{\text{At each cycle, we commit to 5 (base) field elements:}}$

- pc (program counter)
- fp (frame pointer)
- addr_A, addr_B, 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):

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

2.6.4 Transition constraints

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

We define the following quantities:

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

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:

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

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

When there is no jump:

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

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

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

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

3 Annex A: A batched version of logup*

We use the same notations as in the logup* paper [8], 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 α 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 $X \leftarrow eq_{r_1} + \alpha \cdot eq_{r_2}$.

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] S. Setty, S. Angel, T. Gupta, and J. Lee, "Proving the correct execution of concurrent services in zero-knowledge," Cryptology ePrint Archive, Paper 2018/907, 2018. [Online]. Available: https://eprint.iacr.org/2018/907

[8] 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