Minimal zkVM for the Beam Chain (draft 0.2)

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?

The Beam chain initiative involves replacing the BLS signature scheme with a Post-Quantum alternative. One approcah is to use small hash-based signatures (XMSS, 2.5 KiB) see [1] and [2], and to use a hash-based SNARK to handle aggregation. A candidate hash function is Poseidon2 [3].

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 [4]) 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 $\leftarrow$ (**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

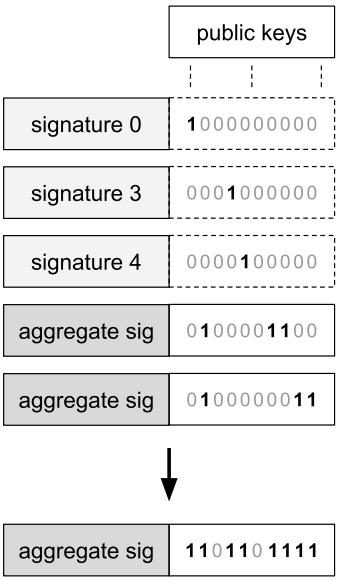
2 Specification (draft)

2.1 Field

2.1.1 Base field

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

Extension of dimension 5 or 6

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

2.2 Memory

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 [6]) or a memory argument based on reordering, which in both case require a range check at each memory operation.

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). We briefly describe how to design a compiler for a high level language without this register:

1. We replace all loops with recursive function.
2. We determine for each function the size its memory footprint (in case of if/else, take the maximum of each block).
3. Each time we enter a function call, we store into the bytecode a hint (something which is not part of the final bytecode verified by the zkVM), to allocate a certain amount of memory, and to place a pointer the allocated frame at a given memory cell (known at compile time, relative to fp). After having stored the current values of pc / fp, and the function's arguments at the beginning of the new frame, we can then jump, to enter the function bytecode, and modify fp with the hinted value. The argument here is that the verifier does not care where the new memory frame will be placed. In practice, the prover that runs the program would need to keep the value of "ap", in order to adequately allocate new memory frames, but there is no need to keep track of it from the versifier's perspective.

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 JUZ (Jump unless zero)

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

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

2.4.4 4 Precompiles

We need two Poseidon2 precompiles: one for the permutation over 16 field elements: POSEIDON_16, and one over 24 field elements: POSEIDON_24.

In order to speed up recursion, we also use precompiles to perform dot products: between two slices of extension field elements: DOT_PRODUCT_EE and between one slice of base field elements and one slice of extension field elements DOT_PRODUCT_BE.

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_EE$((\mathbf{m_8}[a], \dots, \mathbf{m_8}[a+n]), (\mathbf{m_8}[b], \dots, \mathbf{m_8}[b+n]) = \mathbf{m_8}[c]$
- DOT_PRODUCT_BE$((\mathbf{m}[a], \dots, \mathbf{m}[a+n]), (\mathbf{m_8}[b], \dots, \mathbf{m_8}[b+n])) = \mathbf{m_8}[c]$

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 dot product precompiles, for $x = a, b$ or x, 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 AIR for the ISA

2.5.1 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*[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).

2.5.2 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, JUZ, POSEIDON_16, POSEIDON_24, DOT_PRODUCT_EE, DOT_PRODUCT_BE
- One multi-purpose operand: AUX

2.5.3 Execution table

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):

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

2.5.4 Transition constraints

We use transition constraints of degree 4, 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 $\text{DEREF} = 1$, set $\text{flag}_A = 0$, $\text{flag}_C = 1$ and:

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

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

When there is no jump:

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

When $\text{JUZ} = 1$, the condition is represented by ν_A:

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

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

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] 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

[4] 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

[5] 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

[6] 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

[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