Minimal zkVM for the Beam Chain (draft)

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]) would be useful as glue to handle all the logic.

Aggregate / Merge can be unified in a single program, which is the only one the zkVM would have 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

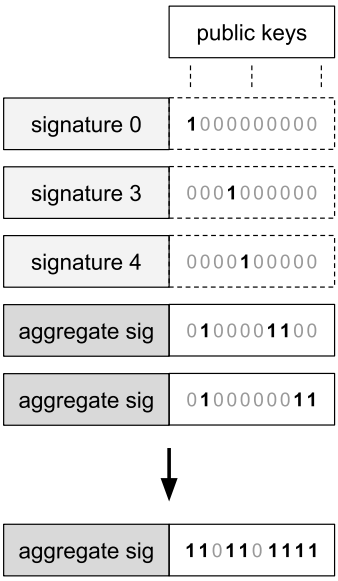
2 Specification (draft)

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)

Drawbacks:

- Small 2-addicity (24)

Note: we can work around this low 2-addicity: In an extension of degree 2^d, the 2-addicity becomes $24 + d$.

2.1.2 Extension field

Extension of degree 8

Why not degree 4 ? Because then it would be impossible to get near 128 bits of security in the "folding operation" of FRI/WHIR [1].

We could get up to 128 bits of security with a degree 5 extension (assuming the conjecture 8.4 of [5] is true with $c_1 = c_2 = 1$), but we would lose the gain in the 2-addicity offered by an extension degree which is a power of two.

A degree 8 extension gives us:

- 128 bits of security in the folding operation of WHIR (the authors of the WHIR paper are confident the proof of [5] for the Johnson bound can be adapted to "mutual" correlated agreement)
- higher 2-addicity (27 instead of 24 for the degree 5 extension)
- the possibility to use Ring Switching ([6] requires a power-of-two extension degree)

In the following, we denote by $\mathbb{F}_p$ the KoalaBear field (base field), and by $\mathbb{F}_q$ its degree 8 extension field.

Note: Another alternative could be: degree 5 extension + WHIR on a domain of size $127 \cdot 2^k$, using mixed radix FFT + Bluestein's algorithm (idea of A. Sanso).

[1] Expect in the unique decoding regime, where proofs are multiple times larger / or with EthStark conjecture

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

The memory hosts:

- bytecode of the program
- public input
- runtime memory

Memory = a vector $\mathbf{m}$ of field elements in $\mathbb{F}_p$ (KoalaBear), with 2 types of access:

- scalar access $i \to \mathbf{m}[i] \in \mathbb{F}_p$
- vector access $i \to \hat{\mathbf{m}}[i] = \mathbf{m}[8i..8(i+1)] \in (\mathbb{F}_p)^8$

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 ISA

TODO: improve, simplify

2.4.1 ADD / MUL

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

$$a = \begin{cases} \text{constant} \\ \mathbf{m}[\text{fp + constant}] \end{cases} \qquad b = \begin{cases} \text{fp} \\ \mathbf{m}[\text{fp + constant'}] \end{cases} \qquad c = \begin{cases} \text{constant"} \\ \mathbf{m}[\text{fp + constant"}] \end{cases} \tag{1}$$

2.4.2 DEREF

$$\mathbf{m}[\mathbf{m}[\text{fp + constant}] + \text{constant'}] = \begin{cases} \text{constant"} \\ \mathbf{m}[\text{fp + constant"}] \\ \text{fp} \end{cases}$$

2.4.3 JUZ (Jump unless zero)

$$\text{condition} = \begin{cases} \text{constant} \\ \mathbf{m}[\text{fp + constant}] \end{cases} \qquad \text{dest} = \begin{cases} \text{constant} \\ \mathbf{m}[\text{fp + constant'}] \end{cases} \qquad \text{next(fp)} = \begin{cases} \text{fp} \\ \mathbf{m}[\text{fp + constant"}] \end{cases} \tag{2}$$

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

2.4.4 Poseidon2$_{16}$ precompile

A precompile for Poseidon2 permutation over 16 field elements. Takes only one parameter: "s" (shift).

Poseidon2$_{16}$(s) $\Leftrightarrow$ Poseidon2($\hat{\mathbf{m}}[\mathbf{m}[\text{fp} + s]], \hat{\mathbf{m}}[\mathbf{m}[\text{fp} + s + 1]]) = \hat{\mathbf{m}}[\mathbf{m}[\text{fp} + s + 2]], \hat{\mathbf{m}}[\mathbf{m}[\text{fp} + s + 3]]$

Where both pairs of 8 field elements are concatenated together. Note the use of the vectorized memory (by chunks of 8 field elements).

2.4.5 Poseidon2$_{24}$ precompile

A similar precompile for Poseidon2 permutation, but over 24 field elements. Operates in a similar manner.

2.4.6 ExtMul precompile?

We may also need a precompile to perform multiplications in the extension field (of degree 8), to improve the speed of recursion. TBD

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 are packed. In the execution trace, this leads to committing to 15 field elements at each instruction (unpacking the flags).

I believe we can significantly reduce the commitments cost using logup*[8]. We can indeed describe each instruction with a dozen (in our case, 13) field elements. In the the execution table, we would only need to commit to the pc column, and all the 13 field elements describing the current instruction being executed can be interpreted as the result an indexed lookup argument (for which logup* would drastically reduce commitment costs).

2.5.2 Instruction Encoding

Each instruction is described by 13 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$_C$
- 7 opcode flags ($\in \{0, 1\}$): ADD, MUL, DEREF_CM, DEREF_FP, JUZ, Poseidon2$_{16}$, Poseidon2$_{24}$

2.5.3 Execution table

- pc (program counter)
- fp (frame pointer)
- jump (non zero when a jump occurs)
- addr_A, addr_B, addr_C
- ν_A, ν_B, ν_C

The following 3 columns can be interpreted as indexed lookups, and thus not be committed thanks to logup* (in practice though, the gains would be more modest than compared to the indexed lookup for 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 (quadratic)

When flag = 1, $\nu_A = \text{operand}_A$, $\nu_B = \text{fp}$, $\nu_C = \text{operand}_C$:

- $\text{flag}_A \cdot (\nu_A - \text{operand}_A) = 0$
- $\text{flag}_B \cdot (\nu_B - \text{fp}) = 0$
- $\text{flag}_C \cdot (\nu_C - \text{operand}_C) = 0$

When flag = 0, $\nu_A = \mathbf{m}[\text{fp} + \text{operand}_A]$, $\nu_B = \mathbf{m}[\text{fp} + \text{operand}_B]$, $\nu_C = \mathbf{m}[\text{fp} + \text{operand}_C]$:

- $(1 - \text{flag}_A) \cdot (\text{address}_A - (\text{fp} + \text{operand}_A)) = 0$ and $(1 - \text{flag}_A) \cdot (\nu_A - \text{value}_A) = 0$
- $(1 - \text{flag}_B) \cdot (\text{address}_B - (\text{fp} + \text{operand}_B)) = 0$ and $(1 - \text{flag}_B) \cdot (\nu_B - \text{value}_B) = 0$
- $(1 - \text{flag}_C) \cdot (\text{address}_C - (\text{fp} + \text{operand}_C)) = 0$ and $(1 - \text{flag}_C) \cdot (\nu_C - \text{value}_C) = 0$

ADD / MUL:

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

To use DEREF, set $\text{flag}_A = 0$, $\text{flag}_B = 1$ and:

$$\mathbf{m}[\mathbf{m}[\text{fp + constant}] + \text{constant'}] = \begin{cases} \text{constant"} & \rightarrow \quad \text{DEREF\_CM} = 1,\ \text{flag}_C = 1 \\ \mathbf{m}[\text{fp + constant"}] & \rightarrow \quad \text{DEREF\_CM} = 1,\ \text{flag}_C = 0 \\ \text{fp} & \rightarrow \quad \text{DEREF\_FP} = 1 \end{cases}$$

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

JUZ:

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

2.5.5 Alternative: cubic constraints

If the PCS cost dominates the PIOP cost, we could alternatively use degree 3 constraints, in which case it's possible to reduce the number of field elements committed per instruction to only 5: pc, fp, addr_A, addr_B, addr_C (again, using logup*).

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] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf, "Proximity gaps for reed-solomon codes," Cryptology ePrint Archive, Paper 2020/654, 2020. [Online]. Available: https://eprint.iacr.org/2020/654

[6] B. E. Diamond and J. Posen, "Polylogarithmic proofs for multilinears over binary towers," 2024. [Online]. Available: https://eprint.iacr.org/2024/504

[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