

The Polygon Zero zkEVM

DRAFT
August 19, 2024

Abstract

We describe the design of Polygon Zero's zkEVM, ...

Contents

1	Introduction	4
2	STARK framework	4
2.1	Cost model	4
2.2	Field selection	4
2.3	Cross-table lookups	5
2.4	Range-checks	6
2.4.1	What to range-check?	6
2.4.2	Lookup Argument	8
2.4.3	Constraints	9
3	Tables	10
3.1	CPU	10
3.1.1	CPU flow	10
3.1.2	CPU columns	11
3.2	Arithmetic	13
3.2.1	Auxiliary columns	15
3.3	Byte Packing	16
3.4	Logic	18
3.5	Memory	19
3.5.1	Virtual memory	20
3.5.2	Timestamps	21
3.5.3	Memory initialization	21
3.5.4	Final memory	22
3.6	Memory continuations	22
3.7	Keccak-f	22
3.7.1	Keccak-f Permutation	22
3.7.2	Columns	24
3.7.3	Constraints	25
3.8	KeccakSponge	26
4	Merkle Patricia Tries	28
4.1	Internal memory format	29
4.1.1	State trie	29
4.1.2	Transaction Trie	30
4.1.3	Receipt Trie	30
4.2	Prover input format	30
4.3	Encoding and Hashing	31

5	CPU logic	32
5.1	Kernel	32
5.2	Simple opcodes VS Syscalls	35
5.3	Privileged instructions	35
5.4	Memory addresses	37
5.5	Stack handling	38
	5.5.1 Top of the stack	38
	5.5.2 Stack length checking	39
5.6	Gas handling	40
	5.6.1 Out of gas errors	40
	5.6.2 Overflow	41
5.7	Exceptions	41
5.8	Linked lists	43

1 Introduction

TODO

2 STARK framework

2.1 Cost model

Our zkEVM is designed for efficient verification by STARKs [1], particularly by an AIR with degree 3 constraints. In this model, the prover bottleneck is typically constructing Merkle trees, particularly constructing the tree containing low-degree extensions of witness polynomials.

2.2 Field selection

Our zkEVM is designed to have its execution traces encoded in a particular prime field \mathbb{F}_p , with $p = 2^{64} - 2^{32} + 1$. A nice property of this field is that it can represent the results of many common u32 operations. For example, (widening) u32 multiplication has a maximum value of $(2^{32} - 1)^2$, which is less than p . In fact a u32 multiply-add has a maximum value of $p - 1$, so the result can be represented with a single field element, although if we were to add a carry in bit, this no longer holds.

This field also enables a very efficient reduction method. Observe that

$$2^{64} \equiv 2^{32} - 1 \pmod{p}$$

and consequently

$$\begin{aligned} 2^{96} &\equiv 2^{32}(2^{32} - 1) \pmod{p} \\ &\equiv 2^{64} - 2^{32} \pmod{p} \\ &\equiv -1 \pmod{p}. \end{aligned}$$

To reduce a 128-bit number n , we first rewrite n as $n_0 + 2^{64}n_1 + 2^{96}n_2$, where n_0 is 64 bits and n_1, n_2 are 32 bits each. Then

$$\begin{aligned} n &\equiv n_0 + 2^{64}n_1 + 2^{96}n_2 \pmod{p} \\ &\equiv n_0 + (2^{32} - 1)n_1 - n_2 \pmod{p} \end{aligned}$$

After computing $(2^{32} - 1)n_1$, which can be done with a shift and subtraction, we add the first two terms, subtracting p if overflow occurs. We then subtract n_2 , adding p if underflow occurs.

At this point we have reduced n to a `u64`. This partial reduction is adequate for most purposes, but if we needed the result in canonical form, we would perform a final conditional subtraction.

2.3 Cross-table lookups

The various STARK tables carry out independent operations, but on shared values. We need to check that the shared values are identical in all the STARKs that require them. This is where cross-table lookups (CTLs) come in handy.

Suppose STARK S_1 requires an operation – say Op – that is carried out by another STARK S_2 . Then S_1 writes the input and output of Op in its own table, and provides the inputs to S_2 . S_2 also writes the inputs and outputs in its rows, and the table’s constraints check that Op is carried out correctly. We then need to ensure that the inputs and outputs are the same in S_1 and S_2 .

In other words, we need to ensure that the rows – reduced to the input and output columns – of S_1 calling Op are permutations of the rows of S_2 that carry out Op . Our CTL protocol is based on `logUp` and is similar to our range-checks.

To prove this, the first step is to only select the rows of interest in S_1 and S_2 , and filter out the rest. Let f^1 be the filter for S_1 and f^2 the filter for S_2 . f^1 and f^2 are constrained to be in $\{0, 1\}$. $f^1 = 1$ (resp. $f^2 = 1$) whenever the row at hand carries out Op in S_1 (resp. in S_2), and 0 otherwise. Let also (α, β) be two random challenges.

The idea is to create subtables S'_1 and S'_2 of S_1 and S_2 respectively, such that $f^1 = 1$ and $f^2 = 1$ for all their rows. The columns in the subtables are limited to the ones whose values must be identical (the inputs and outputs of Op in our example).

Note that for design and constraint reasons, filters are limited to (at most) degree 2 combinations of columns.

Let $\{c^{1,i}\}_{i=1}^m$ be the columns in S'_1 and $\{c^{2,i}\}_{i=1}^m$ be the columns in S'_2 .

The prover defines a “running sum” Z for S'_1 such that:

$$Z_{n-1}^{S_1} = \frac{1}{\sum_{j=0}^{m-1} \alpha^j \cdot c_{n-1}^{1,j} + \beta}$$

$$Z_{i+1}^{S_1} = Z_i^{S_1} + f_i^1 \cdot \frac{1}{\sum_{j=0}^{m-1} \alpha^j \cdot c_i^{1,j} + \beta}$$

The second equation “selects” the terms of interest thanks to f^1 and filters out the rest.

Similarly, the prover constructs a running sum Z^{S_2} for S_2 . Note that Z is computed “upside down”: we start with Z_{n-1} and the final sum is in Z_0 .

On top of the constraints to check that the running sums were correctly constructed, the verifier checks that $Z_0^{S_1} = Z_0^{S_2}$. This ensures that the columns in S'_1 and the columns in S'_2 are permutations of each other.

In other words, the CTL argument is a logUp lookup argument where S'_1 is the looking table, S'_2 is the looked table, and $S'_1 = S'_2$ (all the multiplicities are 1). For more details about logUp, see the next section.

To sum up, for each STARK S , the prover:

1. constructs a running sum Z_i^l for each table looking into S (called looking sums here),
2. constructs a running sum Z^S for S (called looked sum here),
3. sends the final value for each running sum $Z_{i,0}^l$ and Z_0^S to the verifier,
4. sends a commitment to Z_i^l and Z^S to the verifier.

Then, for each STARK S , the verifier:

1. computes the sum $Z = \sum_i Z_{i,0}^l$,
2. checks that $Z = Z_0^S$,
3. checks that each Z_i^l and Z^S was correctly constructed.

2.4 Range-checks

In most cases, tables deal with U256 words, split into 32-bit limbs (to avoid overflowing the field). To prevent a malicious prover from cheating, it is crucial to range-check those limbs.

2.4.1 What to range-check?

One can note that every element that ever appears on the stack has been pushed. Therefore, enforcing a range-check on pushed elements is enough to range-check all elements on the stack. Similarly, all elements in memory must have been written prior, and therefore it is enough to range-check memory writes. However, range-checking the PUSH and MSTORE opcodes is not sufficient.

1. Pushes and memory writes for “MSTORE_32BYTES” are range-checked in “BytePackingStark”, except PUSH operations happening in privileged mode. See 3.1.2.
2. Syscalls, exceptions and prover inputs are range-checked in “ArithmeticStark”.
3. The inputs and outputs of binary and ternary arithmetic operations are range-checked in “ArithmeticStark”.
4. The inputs’ bits of logic operations are checked to be either 1 or 0 in “LogicStark”. Since “LogicStark” only deals with bitwise operations, this is enough to have range-checked outputs as well.
5. The inputs of Keccak operations are range-checked in “KeccakStark”. The output digest is written as bytes in “KeccakStark”. Those bytes are used to reconstruct the associated 32-bit limbs checked against the limbs in “CpuStark”. This implicitly ensures that the output is range-checked.

Note that some operations do not require a range-check:

1. “MSTORE_GENERAL” read the value to write from the stack. Thus, the written value was already range-checked by a previous push.
2. “EQ” reads two – already range-checked – elements on the stack, and checks they are equal. The output is either 0 or 1, and does therefore not need to be checked.
3. “NOT” reads one – already range-checked – element. The result is constrained to be equal to $0xFFFFFFFF - \text{input}$, which implicitly enforces the range check.
4. “PC”: the program counter cannot be greater than 2^{32} in user mode. Indeed, the user code cannot be longer than 2^{32} , and jumps are constrained to be JUMPDESTs. Moreover, in kernel mode, every jump is towards a location within the kernel, and the kernel code is smaller than 2^{32} . These two points implicitly enforce PC ’s range check.
5. “GET_CONTEXT”, “DUP” and “SWAP” all read and push values that were already written in memory. The pushed values were therefore already range-checked.

Range-checks are performed on the range $[0, 2^{16} - 1]$, to limit the trace length.

2.4.2 Lookup Argument

To enforce the range-checks, we leverage [logUp](#), a lookup argument by Ulrich Häbock. Given a looking table $s = (s_1, \dots, s_n)$ and a looked table $t = (t_1, \dots, t_m)$, the goal is to prove that

$$\forall 1 \leq i \leq n, \exists 1 \leq j \leq r \text{ such that } s_i = t_j$$

In our case, $t = (0, \dots, 2^{16} - 1)$ and s is composed of all the columns in each STARK that must be range-checked.

The logUp paper explains that proving the previous assertion is actually equivalent to proving that there exists a sequence l such that:

$$\sum_{i=1}^n \frac{1}{X - s_i} = \sum_{j=1}^r \frac{l_j}{X - t_j}$$

The values of s can be stored in c different columns of length n each. In that case, the equality becomes:

$$\sum_{k=1}^c \sum_{i=1}^n \frac{1}{X - s_i^k} = \sum_{j=1}^r \frac{l_j}{X - t_j}$$

The ‘multiplicity’ m_i of value t_i is defined as the number of times t_i appears in s . In other words:

$$m_i = |s_j \in s; s_j = t_i|$$

Multiplicities provide a valid sequence of values in the previously stated equation. Thus, if we store the multiplicities, and are provided with a challenge α , we can prove the lookup argument by ensuring:

$$\sum_{k=1}^c \sum_{i=1}^n \frac{1}{\alpha - s_i^k} = \sum_{j=1}^r \frac{m_j}{\alpha - t_j}$$

However, the equation is too high degree. To circumvent this issue, Häbock suggests providing helper columns h_i and d such that at a given row i :

$$h_i^k = \frac{1}{\alpha + s_i^k} \forall 1 \leq k \leq c$$

$$d_i = \frac{1}{\alpha + t_i}$$

The h helper columns can be batched together to save columns. We can batch at most `constraint_degree` $- 1$ helper functions together. In our case, we batch them 2 by 2. At row i , we now have:

$$h_i^k = \frac{1}{\alpha + s_i^{2k}} + \frac{1}{\alpha + s_i^{2k+1}} \forall 1 \leq k \leq c/2$$

If c is odd, then we have one extra helper column:

$$h_i^{c/2+1} = \frac{1}{\alpha + s_i^c}$$

For clarity, we will assume that c is even in what follows.

Let g be a generator of a subgroup of order n . We extrapolate h, m and d to get polynomials such that, for $f \in \{h^k, m, g\}$: $f(g^i) = f_i$. We can define the following polynomial:

$$Z(x) := \sum_{i=1}^n \left[\sum_{k=1}^{c/2} h^k(x) - m(x) * d(x) \right]$$

2.4.3 Constraints

With these definitions and a challenge α , we can finally check that the assertion holds with the following constraints:

$$\begin{aligned} Z(1) &= 0 \\ Z(g\alpha) &= Z(\alpha) + \sum_{k=1}^{c/2} h^k(\alpha) - m(\alpha)d(\alpha) \end{aligned}$$

These ensure that We also need to ensure that h^k is well constructed for all $1 \leq k \leq c/2$:

$$h(\alpha)^k \cdot (\alpha + s_{2k}) \cdot (\alpha + s_{2k+1}) = (\alpha + s_{2k}) + (\alpha + s_{2k+1})$$

Note: if c is odd, we have one unbatched helper column $h^{c/2+1}$ for which we need a last constraint:

$$h(\alpha)^{c/2+1} \cdot (\alpha + s_c) = 1$$

Finally, the verifier needs to ensure that the table t was also correctly computed. In each STARK, t is computed starting from 0 and adding at most 1 at each row. This construction is constrained as follows:

1. $t(1) = 0$
2. $(t(g^{i+1}) - t(g^i)) \cdot ((t(g^{i+1}) - t(g^i)) - 1) = 0$
3. $t(g^{n-1}) = 2^{16} - 1$

3 Tables

3.1 CPU

The CPU is the central component of the zkEVM. Like any CPU, it reads instructions, executes them and modifies the state (registers and the memory) accordingly. The constraining of some complex instructions (e.g. Keccak hashing) is delegated to other tables. This section will only briefly present the CPU and its columns. Details about the CPU logic will be provided later.

3.1.1 CPU flow

An execution run can be decomposed into two distinct parts:

- **CPU cycles:** The bulk of the execution. In each row, the CPU reads the current code at the program counter (PC) address, and executes it. The current code can be the kernel code, or whichever code is being executed in the current context (transaction code or contract code). Executing an instruction consists in modifying the registers, possibly performing some memory operations, and updating the PC.
- **Padding:** At the end of the execution, we need to pad the length of the CPU trace to the next power of two. When the program counter reaches the special halting label in the kernel, execution halts. Constraints ensure that every subsequent row is a padding row and that execution cannot resume.

In the CPU cycles phase, the CPU can switch between different contexts, which correspond to the different environments of the possible calls. Context 0 is the kernel itself, which handles initialization (input processing, transaction parsing, transaction trie updating...) and termination (receipt creation, final trie checks...) before and after executing the transaction. Subsequent contexts are created when executing user code (transaction or contract code). In a non-zero user context, syscalls may be executed, which are specific instructions written in the kernel. They don't change the context but change the code context, which is where the instructions are read from.

Continuations A full run of the zkEVM consists in initializing the zkEVM with the input state, executing a certain number of transactions, and then validating the output state. However, for performance reasons, a run is split in multiple segments of at most `MAX_CPU_CYCLES` cycles, which can be proven individually. Continuations ensure that the segments are part of the same run and guarantees that the state at the end of a segment is equal to the state at the beginning of the next.

The state to propagate from one segment to another contains some of the zkEVM registers plus the current memory. These registers are stored in memory as dedicated global metadata, and the memory to propagate is stored in two STARK tables: `MemBefore` and `MemAfter`. To check the consistency of the memory, the Merkle cap of the previous `MemAfter` is compared to the Merkle cap of the next `MemBefore`.

3.1.2 CPU columns

Registers:

- **context**: Indicates which context we are in. 0 for the kernel, and a positive integer for every user context. Incremented by 1 at every call.
- **code_context**: Indicates in which context the code to execute resides. It's equal to **context** in user mode, but is always 0 in kernel mode.
- **program_counter**: The address of the instruction to be read and executed.
- **stack_len**: The current length of the stack.
- **is_kernel_mode**: Boolean indicating whether we are in kernel (i.e. privileged) mode. This means we are executing kernel code, and we have access to privileged instructions.
- **gas**: The current amount of gas used in the current context. It is eventually checked to be below the current gas limit. Must fit in 32 bits.
- **clock**: Monotonic counter which starts at 0 and is incremented by 1 at each row. Used to enforce correct ordering of memory accesses.
- **opcode_bits**: 8 boolean columns, which are the bit decomposition of the opcode being read at the current PC.

Operation flags: Boolean flags. During CPU cycles phase, each row executes a single instruction, which sets one and only one operation flag. No flag is set during padding. The decoding constraints ensure that the flag set corresponds to the opcode being read. There isn't a 1-to-1 correspondence between instructions and flags. For efficiency, the same flag can be set by different, unrelated instructions (e.g. `eq_iszero`, which represents the `EQ` and the `ISZERO` instructions). When there is a need to differentiate them in constraints, we filter them with their respective opcode: since the first bit of `EQ`'s opcode (resp. `ISZERO`'s opcode) is 0 (resp. 1), we can filter a constraint for an `EQ` instruction with `eq_iszero * (1 - opcode_bits[0])` (resp. `eq_iszero * opcode_bits[0]`).

Memory columns: The CPU interacts with the EVM memory via its memory channels. At each row, a memory channel can execute a write, a read, or be disabled. A full memory channel is composed of:

- **used:** Boolean flag. If it's set to 1, a memory operation is executed in this channel at this row. If it's set to 0, no operation is done but its columns might be reused for other purposes.
- **is_read:** Boolean flag indicating if a memory operation is a read or a write.
- **3 address columns.** A memory address is made of three parts: `context`, `segment` and `virtual`.
- **8 value columns.** EVM words are 256 bits long, and they are broken down in 8 32-bit limbs.

The last memory channel is a partial channel: it doesn't have its own `value` columns and shares them with the first full memory channel. This allows us to save eight columns.

General columns: There are 8 shared general columns. Depending on the instruction, they are used differently:

- **Exceptions:** When raising an exception, the first three general columns are the bit decomposition of the exception code. They are used to jump to the correct exception handler.
- **Logic:** For `EQ`, and `ISZERO` operations, it's easy to check that the result is 1 if `input0` and `input1` are equal. It's more difficult to prove that, if

the result is 0, the inputs are actually unequal. To prove it, each general column contains the modular inverse of $(\text{input0}_i - \text{input1}_i)$ for each limb i (or 0 if the limbs are equal). Then the quantity $\text{general}_i * (\text{input0}_i - \text{input1}_i)$ will be 1 if and only if general_i is indeed the modular inverse, which is only possible if the difference is non-zero.

- **Jumps:** For jumps, we use the first two columns: `should_jump` and `cond_sum_pinv`. `should_jump` conditions whether the EVM should jump: it's 1 for a JUMP, and `condition` $\neq 0$ for a JUMPI. To check if the condition is actually non-zero for a JUMPI, `cond_sum_pinv` stores the modular inverse of `condition` (or 0 if it's zero).
- **Shift:** For shifts, the logic differs depending on whether the displacement is lower than 2^{32} , i.e. if it fits in a single value limb. To check if this is not the case, we must check that at least one of the seven high limbs is not zero. The general column `high_limb_sum_inv` holds the modular inverse of the sum of the seven high limbs, and is used to check it's non-zero like the previous cases. Contrary to the logic operations, we do not need to check limbs individually: each limb has been range-checked to 32 bits, meaning that it's not possible for the sum to overflow and be zero if some of the limbs are non-zero.
- **Stack:** `stack_inv`, `stack_inv_aux` and `stack_inv_aux_2` are used by popping-only (resp. pushing-only) instructions to check if the stack is empty after (resp. was empty before) the instruction. `stack_len_bounds_aux` is used to check that the stack doesn't overflow in user mode. We use the last four columns to prevent conflicts with the other general columns. See 5.5 for more details.
- **Push:** `is_not_kernel` is used to skip range-checking the output of a PUSH operation when we are in privileged mode, as the kernel code is known and trusted.
- **Context pruning:** When SET_CONTEXT is called to return to a parent context, this makes the current context stale. The kernel indicates it by setting one general column to 1. For more details about context pruning, see 3.5.4.

3.2 Arithmetic

Each row of the arithmetic table corresponds to a binary or ternary arithmetic operation. Each of these operations has an associated flag f_{op} in the table, such

that $f_{\text{op}} = 1$ whenever the operation is `op` and 0 otherwise. The full list of operations carried out by the table is as follows:

Binary operations:

- basic operations: “add”, “mul”, “sub” and “div”,
- comparisons: “lt” and “gt”,
- shifts: “shr” and “shl”,
- “byte”: given x_1, x_2 , returns the x_1 -th “byte” in x_2 ,
- modular operations: “mod”, “AddFp254”, “MulFp254” and “SubFp254”,
- range-check: no operation is performed, as this is only used to range-check the input and output limbs in the range $[0, 2^{16} - 1]$.

For ‘mod’, the second input is the modulus. “AddFp254”, “MulFp254” and “SubFp254” are modular operations modulo “Fp254” – the prime for the BN curve’s base field.

Ternary operations: There are three ternary operations: modular addition “AddMod”, modular multiplication “MulMod” and modular subtraction “SubMod”.

Besides the flags, the arithmetic table needs to store the inputs, output and some auxiliary values necessary to constraints. The input and output values are range-checked to ensure their canonical representation. Inputs are 256-bits words. To avoid having too large a range-check, inputs are therefore split into sixteen 16-bits limbs, and range-checked in the range $[0, 2^{16} - 1]$.

Overall, the table comprises the following columns:

- 17 columns for the operation flags f_{op} ,
- 1 column op containing the opcode,
- 16 columns for the 16-bit limbs $x_{0,i}$ of the first input x_0 ,
- 16 columns for the 16-bit limbs $x_{1,i}$ of the second input x_1 ,
- 16 columns for the 16-bit limbs $x_{2,i}$ of the third input x_2 ,
- 16 columns for the 16-bit limbs r_i of the output r ,

- 32 columns for auxiliary values aux_i ,
- 1 column `range_counter` containing values in the range $[0, 2^{16} - 1]$, for the range-check,
- 1 column storing the frequency of appearance of each value in the range $[0, 2^{16} - 1]$.

Note on *op*: The opcode column is only used for range-checks. For optimization purposes, we check all arithmetic operations against the `cpu` table together. To ensure correctness, we also check that the operation’s opcode corresponds to its behavior. But range-check is not associated to a unique operation: any operation in the `cpu` table might require its values to be checked. Thus, the arithmetic table cannot know its opcode in advance: it needs to store the value provided by the `cpu` table.

3.2.1 Auxiliary columns

The way auxiliary values are leveraged to efficiently check correctness is not trivial, but it is explained in detail in each dedicated file. Overall, five files explain the implementations of the various checks. Refer to:

1. “mul.rs” for details on multiplications.
2. “addcy.rs” for details on addition, subtraction, “lt” and “gt”.
3. “modular.rs” for details on how modular operations are checked. Note that even though “div” and “mod” are generated and checked in a separate file, they leverage the logic for modular operations described in “modular.rs”.
4. “byte” for details on how “byte” is checked.
5. “shift.rs” for details on how shifts are checked.

Note on “lt” and “gt”: For “lt” and “gt”, auxiliary columns hold the difference d between the two inputs x_1, x_2 . We can then treat them similarly to subtractions by ensuring that $x_1 - x_2 = d$ for “lt” and $x_2 - x_1 = d$ for “gt”. An auxiliary column `cy` is used for the carry in additions and subtractions. In the comparisons case, it holds the overflow flag. Contrary to subtractions, the output of “lt” and “gt” operations is not d but `cy`.

Note on “div”: It might be unclear why “div” and “mod” are dealt with in the same file.

Given numerator and denominator n, d , we compute, like for other modular operations, the quotient q and remainder **rem**:

$$\text{div}(x_1, x_2) = q * x_2 + \text{rem}$$

. We then set the associated auxiliary columns to **rem** and the output to q .

This is why “div” is essentially a modulo operation, and can be addressed in almost the same way as “mod”. The only difference is that in the “mod” case, the output is **rem** and the auxiliary value is q .

Note on shifts: “shr” and “shl” are internally constrained as “div” and “mul” respectively with shifted operands. Indeed, given inputs s, x , the output should be $x \gg s$ for “shr” (resp. $x \ll s$ for “shl”). Since shifts are binary operations, we can use the third input columns to store $s_{\text{shifted}} = 1 \ll s$. Then, we can use the “div” logic (resp. “mul” logic) to ensure that the output is $\frac{x}{s_{\text{shifted}}}$ (resp. $x * s_{\text{shifted}}$).

3.3 Byte Packing

The BytePacking STARK module is used for reading and writing non-empty byte sequences of length at most 32 to memory. The “packing” term highlights that reading a sequence in memory will pack the bytes into an EVM word (i.e. U256), while the “unpacking” operation consists in breaking down an EVM word into its byte sequence and writing it to memory.

This allows faster memory copies between two memory locations, as well as faster memory reset (see [memcpy.asm](#) and [memset.asm](#) modules).

The ‘BytePackingStark’ table has one row per packing/unpacking operation.

Each row contains the following columns:

1. 5 columns containing information on the initial memory address from which the sequence starts (namely a flag differentiating read and write operations, address context, segment and offset values, as well as timestamp),
2. 32 columns b_i indicating the length of the byte sequence ($b_i = 1$ if the length is $i + 1$, and $b_i = 0$ otherwise),
3. 32 columns v_i indicating the values of the bytes that have been read or written during a sequence,

4. 2 columns r_i needed for range-checking the byte values.

Notes on columns generation: Whenever a byte unpacking operation is called, the value `val` is read from the stack, but because the EVM and the STARKs use different endianness, we need to convert `val` to a little-endian byte sequence. Only then do we resize it to the appropriate length, and prune extra zeros and higher bytes in the process. Finally, we reverse the byte order and write this new sequence into the v_i columns of the table.

Whenever the operation is a byte packing, the bytes are read one by one from memory and stored in the v_i columns of the BytePackingStark table.

Note that because of the different endianness on the memory and EVM sides, we write bytes starting with the last one.

The b_i columns hold a boolean value. $b_i = 1$ whenever we are currently reading or writing the i -th element in the byte sequence. $b_i = 0$ otherwise.

Cross-table lookups: The read or written bytes need to be checked against both the cpu and the memory tables. Whenever we call `MSTORE_32BYTES`, `MLOAD_32BYTES` or `PUSH` on the cpu side, we make use of ‘BytePackingStark’ to make sure we are carrying out the correct operation on the correct values. For this, we check that the following values correspond:

1. the address (comprising the context, the segment, and the virtual address),
2. the length of the byte sequence,
3. the timestamp,
4. the value (either written to or read from the stack)

The address here corresponds to the address of the first byte.

On the other hand, we need to make sure that the read and write operations correspond to the values read or stored on the memory side. We therefore need a CTL for each byte, checking that the following values are identical in ‘MemoryStark’ and ‘BytePackingStark’:

1. a flag indicating whether the operation is a read or a write,
2. the address (context, segment and virtual address),
3. the byte (followed by 0s to make sure the memory address contains a byte and not a U256 word),

4. the timestamp

Note that the virtual address has to be recomputed based on the length of the sequence of bytes. The virtual address for the i -th byte is written as:

$$\text{virt} + \sum_{j=0}^{31} b_j * j - i$$

where $\sum_{j=0}^{31} b_j * j$ is equal to `sequence_length - 1`.

Note on range-check: Range-checking is necessary whenever we do a memory unpacking operation that will write values to memory. These values are constrained by the range-check to be 8-bit values, i.e. fitting between 0 and 255 included. While range-checking values read from memory is not necessary, because we use the same `byte_values` columns for both read and write operations, this extra condition is enforced throughout the whole trace regardless of the operation type.

3.4 Logic

Each row of the logic table corresponds to one bitwise logic operation: either AND, OR or XOR. Each input for these operations is represented as 256 bits, while the output is stored as eight 32-bit limbs.

Each row therefore contains the following columns:

1. f_{and} , an “is and” flag, which should be 1 for an OR operation and 0 otherwise,
2. f_{or} , an “is or” flag, which should be 1 for an OR operation and 0 otherwise,
3. f_{xor} , an “is xor” flag, which should be 1 for a XOR operation and 0 otherwise,
4. 256 columns $x_{1,i}$ for the bits of the first input x_1 ,
5. 256 columns $x_{2,i}$ for the bits of the second input x_2 ,
6. 8 columns r_i for the 32-bit limbs of the output r .

Note that we need all three flags because we need to be able to distinguish between an operation row and a padding row – where all flags are set to 0.

The subdivision into bits is required for the two inputs as the table carries out bitwise operations. The result, on the other hand, is represented in 32-bit limbs since we do not need individual bits and can therefore save the remaining 248 columns. Moreover, the output is checked against the cpu, which stores values in the same way.

3.5 Memory

For simplicity, let’s treat addresses and values as individual field elements. The generalization to multi-element addresses and values is straightforward.

Each row of the memory table corresponds to a single memory operation (a read or a write), and contains the following columns:

1. a , the target address
2. r , an “is read” flag, which should be 1 for a read or 0 for a write
3. v , the value being read or written
4. τ , the timestamp of the operation

The memory table should be ordered by (a, τ) . Note that the correctness of the memory could be checked as follows:

1. Verify the ordering by checking that $(a_i, \tau_i) \leq (a_{i+1}, \tau_{i+1})$ for each consecutive pair.
2. Enumerate the purportedly-ordered log while tracking the “current” value of v .
 - (a) Upon observing an address which doesn’t match that of the previous row, if the address is zero-initialized and if the operation is a read, check that $v = 0$.
 - (b) Upon observing a write, don’t constrain v .
 - (c) Upon observing a read at timestamp τ_i which isn’t the first operation at this address, check that $v_i = v_{i-1}$.

The ordering check is slightly involved since we are comparing multiple columns. To facilitate this, we add an additional column e , where the prover

can indicate whether two consecutive addresses changed. An honest prover will set

$$e_i \leftarrow \begin{cases} 1 & \text{if } a_i \neq a_{i+1}, \\ 0 & \text{otherwise.} \end{cases}$$

We also introduce a range-check column c , which should hold:

$$c_i \leftarrow \begin{cases} a_{i+1} - a_i - 1 & \text{if } e_i = 1, \\ \tau_{i+1} - \tau_i & \text{otherwise.} \end{cases}$$

The extra -1 ensures that the address actually changed if $e_i = 1$. We then impose the following transition constraints:

1. $e_i(e_i - 1) = 0$,
2. $(1 - e_i)(a_{i+1} - a_i) = 0$,
3. $c_i < 2^{32}$.

The third constraint emulates a comparison between two addresses or timestamps by bounding their difference; this assumes that all addresses and timestamps fit in 32 bits and that the field is larger than that.

3.5.1 Virtual memory

In the EVM, each contract call has its own address space. Within that address space, there are separate segments for code, main memory, stack memory, calldata, and returndata. Thus each address actually has three components:

1. an execution context, representing a contract call,
2. a segment ID, used to separate code, main memory, and so forth, and so on
3. a virtual address.

The comparisons now involve several columns, which requires some minor adaptations to the technique described above; we will leave these as an exercise to the reader.

Note that an additional constraint check is required: whenever we change the context or the segment, the virtual address must be range-checked to 2^{32} . Without this check, addresses could start at -1 (i.e. $p - 2$) and then increase properly.

3.5.2 Timestamps

Memory operations are sorted by address a and timestamp τ . For a memory operation in the CPU, we have:

$$\tau = \text{NUM_CHANNELS} \times \text{cycle} + \text{channel}.$$

Since a memory channel can only hold at most one memory operation, every CPU memory operation's timestamp is unique.

Note that it doesn't mean that all memory operations have unique timestamps. There are two exceptions:

- Before the CPU cycles, we preinitialize the memory with the flashed state stored in the MemBefore table and we write some global metadata. These operations are done at timestamp $\tau = 0$.
- Some tables other than CPU can generate memory operations, like KeccakSponge. When this happens, these operations all have the timestamp of the CPU row of the instruction which invoked the table (for KeccakSponge, KECCAK_GENERAL).

3.5.3 Memory initialization

By default, all memory is zero-initialized. However, to save numerous writes, we allow some specific segments to be initialized with arbitrary values.

- The code segment (segment 0) is either part of the initial memory for the kernel (context 0), or is initialized with externally-provided account code, then checked against the account code hash. In non-zero contexts, if the code is meant to be executed, there is a soundness concern: if the code is malformed and ends with an incomplete PUSH, then the missing bytes must be 0 accordingly to the Ethereum specs. To prevent the issue, we manually write 33 zeros (at most 32 bytes for the PUSH argument, and an extra one for the post-PUSH PC value).
- The “TrieData” segment is initialized with the input tries. The stored tries are hashed and checked against the provided initial hash. Note that the length of the segment and the pointers – within the “TrieData” segment – for the three tries are provided as prover inputs. The length is then checked against a value computed when hashing the tries.

3.5.4 Final memory

The final value of each cell of the memory must be propagated to the MemAfter table. Since memory operations are ordered by address and by timestamps, this is easy to do: the last value of an address is the value of the last row touching this address. In other words, we propagate values of rows before the address changes.

Context pruning We can observe that whenever we return from a context (e.g. with a RETURN opcode, from an exception...), we will never access it again and all its memory is now stale. We make use of this fact to prune stale contexts and exclude them from MemAfter.

3.6 Memory continuations

The MemBefore (resp. MemAfter) table holds the content of the memory before (resp. after) the execution of the current segment. For consistency, the MemAfter trace of a segment must be identical to the MemAfter trace of the next segment. Each row of these tables contains:

1. a , the memory cell address,
2. v , the initial value of the cell.

The tables should be ordered by (a, τ) . Since they only hold values, there are no constraints between the rows.

A CTL copies all of the MemBefore values in the memory trace as reads, at timestamp $\tau = 0$. Another CTL copies the final values from memory to MemAfter. For more details on which values are propagated, consult [3.5.4](#).

3.7 Keccak-f

This table computes the Keccak-f[1600] permutation.

3.7.1 Keccak-f Permutation

To explain how this table is structured, we first need to detail how the permutation is computed. [This page](#) gives a pseudo-code for the permutation. Our implementation differs slightly – but remains equivalent – for optimization and constraint degree reasons.

Let:

- S be the sponge width ($S = 25$ in our case)
- NUM_ROUNDS be the number of Keccak rounds ($\text{NUM_ROUNDS} = 24$)
- RC a vector of round constants of size NUM_ROUNDS
- I be the input of the permutation, comprised of S 64-bit elements

The first step is to reshape I into a 5×5 matrix. We initialize the state A of the sponge with I :

$$A[x, y] := I[x, y] \quad \forall x, y \in \{0..4\}$$

We store A in the table, and subdivide each 64-bit element into two 32-bit limbs. Then, for each round i , we proceed as follows:

1. First, we define $C[x] := \text{xor}_{i=0}^4 A[x, i]$. We store C as bits in the table. This is because we need to apply a rotation on its elements' bits and carry out `xor` operations in the next step.
2. Then, we store a second vector C' in bits, such that:

$$C'[x, z] = C[x, z] \text{ xor } C[x - 1, z] \text{ xor } C[x + 1, z - 1]$$

.

3. We then need to store the updated value of A :

$$A'[x, y] = A[x, y] \text{ xor } C[x, y] \text{ xor } C'[x, y]$$

Note that this is equivalent to the equation in the official Keccak-f description:

$$A'[x, y] = A[x, y] \text{ xor } C[x - 1, z] \text{ xor } C[x + 1, z - 1]$$

.

4. The previous three points correspond to the θ step in Keccak-f. We can now move on to the ρ and π steps. These steps are written as:

$$B[y, 2 \times x + 3 \times y] := \text{rot}(A'[x, y], r[x, y])$$

where $\text{rot}(\mathbf{a}, \mathbf{s})$ is the bitwise cyclic shift operation, and r is the matrix of rotation offsets. We do not need to store B : B 's bits are only a permutation of A 's bits.

5. The χ step updates the state once again, and we store the new values:

$$A''[x, y] := B[x, y] \text{ xor } (\text{not } B[x + 1, y] \text{ and } B[x + 2, y])$$

Because of the way we carry out constraints (as explained below), we do not need to store the individual bits for A'' : we only need the 32-bit limbs.

6. The final step, ι , consists in updating the first element of the state as follows:

$$A'''[0, 0] = A''[0, 0] \text{ xor } RC[i]$$

where

$$A'''[x, y] = A''[x, y] \forall (x, y) \neq (0, 0)$$

Since only the first element is updated, we only need to store $A'''[0, 0]$ of this updated state. The remaining elements are fetched from A'' . However, because of the bitwise **xor** operation, we do need columns for the bits of $A''[0, 0]$.

Note that all permutation elements are 64-bit long. But they are stored as 32-bit limbs so that we do not overflow the field.

It is also important to note that all bitwise logic operations (**xor** , **not** and **and**) are checked in this table. This is why we need to store the bits of most elements. The logic table can only carry out eight 32-bit logic operations per row. Thus, leveraging it here would drastically increase the number of logic rows, and incur too much overhead in proving time.

3.7.2 Columns

Using the notations from the previous section, we can now list the columns in the table:

1. $\text{NUM_ROUNDS} = 24$ columns c_i to determine which round is currently being computed. $c_i = 1$ when we are in the i -th round, and 0 otherwise. These columns' purpose is to ensure that the correct round constants are used at each round.
2. 1 column t which stores the timestamp at which the Keccak operation was called in the cpu. This column enables us to ensure that inputs and outputs are consistent between the cpu, keccak-sponge and keccak-f tables.

3. $5 \times 5 \times 2 = 50$ columns to store the elements of A . As a reminder, each 64-bit element is divided into two 32-bit limbs, and A comprises $S = 25$ elements.
4. $5 \times 64 = 320$ columns to store the bits of the vector C .
5. $5 \times 64 = 320$ columns to store the bits of the vector C' .
6. $5 \times 5 \times 64 = 1600$ columns to store the bits of A' .
7. $5 \times 5 \times 2 = 50$ columns to store the 32-bit limbs of A'' .
8. 64 columns to store the bits of $A''[0, 0]$.
9. 2 columns to store the two limbs of $A'''[0, 0]$.

In total, this table comprises 2,431 columns.

3.7.3 Constraints

Some constraints checking that the elements are computed correctly are not straightforward. Let us detail them here.

First, it is important to highlight the fact that a **xor** between two elements is of degree 2. Indeed, for $x \text{ xor } y$, the constraint is $x + y - 2 \times x \times y$, which is of degree 2. This implies that a **xor** between 3 elements is of degree 3, which is the maximal constraint degree for our STARKs.

We can check that $C'[x, z] = C[x, z] \text{ xor } C[x - 1, z] \text{ xor } C[x + 1, z - 1]$. However, we cannot directly check that $C[x] = \text{xor}_{i=0}^4 A[x, i]$, as it would be a degree 5 constraint. Instead, we use C' for this constraint. We see that:

$$\text{xor}_{i=0}^4 A'[x, i, z] = C'[x, z]$$

This implies that the difference $d = \sum_{i=0}^4 A'[x, i, z] - C'[x, z]$ is either 0, 2 or 4. We can therefore enforce the following degree 3 constraint instead:

$$d \times (d - 2) \times (d - 4) = 0$$

Additionally, we have to check that A' is well constructed. We know that A' should be such that $A'[x, y, z] = A[x, y, z] \text{ xor } C[x, z] \text{ xor } C'[x, z]$. Since we do not have the bits of A elements but the bits of A' elements, we check the equivalent degree 3 constraint:

$$A[x, y, z] = A'[x, y, z] \text{ xor } C[x, z] \text{ xor } C'[x, z]$$

Finally, the constraints for the remaining elements, A'' and A''' are straightforward: A'' is a three-element bitwise `xor` where all bits involved are already stored and $A'''[0, 0]$ is the output of a simple bitwise `xor` with a round constant.

3.8 KeccakSponge

This table computes the Keccak256 hash, a sponge-based hash built on top of the Keccak-f[1600] permutation. An instance of KeccakSponge takes as input a Memory address a , a length l , and computes the Keccak256 digest of the memory segment starting at a and of size l . An instance can span many rows, each individual row being a single call to the Keccak table. Note that all the read elements must be bytes; the proof will be unverifiable if this is not the case. Following the Keccak specifications, the input string is padded to the next multiple of 136 bytes. Each row contains the following columns:

- Read bytes:
 - 3 address columns: `context`, `segment` and the offset `virt` of a .
 - `timestamp`: the timestamp which will be used for all memory reads of this instance.
 - `already_absorbed_bytes`: keeps track of how many bytes have been hashed in the current instance. At the end of an instance, we should have absorbed l bytes in total.
 - `KECCAK_RATE_BYTES block_bytes` columns: the bytes being absorbed at this row. They are read from memory and will be XORed to the rate part of the current state.
- Input columns:
 - `KECCAK_RATE_U32S original_rate_u32s` columns: hold the rate part of the state before XORing it with `block_bytes`. At the beginning of an instance, they are initialized with 0.
 - `KECCAK_RATE_U32S xored_rate_u32s` columns: hold the original rate XORed with `block_bytes`.
 - `KECCAK_CAPACITY_U32S original_capacity_u32s` columns: hold the capacity part of the state before applying the Keccak permutation.
- Output columns:

- KECCAK_DIGEST_BYTES `updated_digest_state_bytes` columns: the beginning of the output state after applying the Keccak permutation. At the last row of an instance, they hold the computed hash. They are decomposed in bytes for endianness reasons.
- KECCAK_WIDTH_MINUS_DIGEST_U32S `partial_updated_state_u32s` columns: the rest of the output state. They are discarded for the final digest, but are used between instance rows.
- Helper columns:
 - `is_full_input_block`: indicates if the current row has a full input block, i.e. `block_bytes` contains only bytes read from memory and no padding bytes.
 - KECCAK_RATE_BYTES `is_final_input_len` columns: in the final row of an instance, indicate where the final read byte is. If the i -th column is set to 1, it means that all bytes after the i -th are padding bytes. In a full input block, all columns are set to 0.

For each instance, constraints ensure that:

- at each row:
 - `is_full_input_block` and `is_final_input_len` columns are all binary.
 - Only one column in `is_full_input_block` and `is_final_input_len` is set to 1.
 - `xored_rate_u32s` is `original_rate_u32s` XOR `block_bytes`.
 - The CTL with Keccak ensures that (`updated_digest_state_bytes` columns, `partial_updated_state_u32s`) is the Keccak permutation output of (`xored_rate_u32s`, `original_capacity_u32s`).
- at the first row:
 - `original_rate_u32s` is all 0.
 - `already_absorbed_bytes` is 0.
- at each full input row (i.e. `is_full_input_block` is 1, all `is_final_input_len` columns are 0):
 - `context`, `segment`, `virt` and `timestamp` are unchanged in the next row.

- Next `already_absorbed_bytes` is current `already_absorbed_bytes` + `KECCAK_RATE_BYTES`.
- Next (`original_rate_u32s`, `original_capacity_u32s`) is current (`updated_digest_state_bytes` columns, `partial_updated_state_u32s`).
- The CTL with Memory ensures that `block_bytes` is filled with contiguous memory elements [`a + already_absorbed_bytes`, `a + already_absorbed_bytes + KECCAK_RATE_BYTES - 1`]
- at the final row (i.e. `is_full_input_block` is 0, `is_final_input_len`'s i -th column is 1 for a certain i , the rest are 0):
 - The CTL with Memory ensures that `block_bytes` is filled with contiguous memory elements [`a + already_absorbed_bytes`, `a + already_absorbed_bytes + i - 1`]. The rest are padding bytes.
 - The CTL with CPU ensures that `context`, `segment`, `virt` and `timestamp` match the `KECCAK_GENERAL` call.
 - The CTL with CPU ensures that $l = \text{already_absorbed_bytes} + i$.
 - The CTL with CPU ensures that `updated_digest_state_bytes` is the output of the `KECCAK_GENERAL` call.

The trace is padded to the next power of two with dummy rows, whose `is_full_input_block` and `is_final_input_len` columns are all 0.

4 Merkle Patricia Tries

The *EVM World state* is a representation of the different accounts at a particular time, as well as the last processed transactions together with their receipts. The world state is represented using *Merkle Patricia Tries* (MPTs) [2, App. D], and there are three different tries: the state trie, the transaction trie and the receipt trie.

For each transaction we need to show that the prover knows preimages of the hashed initial and final EVM states. When the kernel starts execution, it stores these three tries within the `Segment::TrieData` segment. The prover loads the initial tries from the inputs into memory. Subsequently, the tries are modified during transaction execution, inserting new nodes or deleting existing nodes.

An MPT is composed of five different nodes: branch, extension, leaf, empty and digest nodes. Branch and leaf nodes might contain a payload whose format

depends on the particular trie. The nodes are encoded, primarily using RLP encoding and Hex-prefix encoding (see [2] App. B and C, respectively). The resulting encoding is then hashed, following a strategy similar to that of normal Merkle trees, to generate the trie hashes.

Insertion and deletion is performed in the same way as other MPTs implementations. The only difference is for inserting extension nodes where we create a new node with the new data, instead of modifying the existing one. In the rest of this section we describe how the MPTs are represented in memory, how they are given as input, and how MPTs are hashed.

4.1 Internal memory format

The tries are stored in kernel memory, specifically in the `Segment:TrieData` segment. Each node type is stored as

1. An empty node is encoded as (`MPT_NODE_EMPTY`).
2. A branch node is encoded as (`MPT_NODE_BRANCH`, c_1, \dots, c_{16}, v), where each c_i is a pointer to a child node, and v is a pointer to a value. If a branch node has no associated value, then $v = 0$, i.e. the null pointer.
3. An extension node is encoded as (`MPT_NODE_EXTENSION`, k, c), k represents the part of the key associated with this extension, and is encoded as a 2-tuple (`packed_nibbles`, `num_nibbles`). c is a pointer to a child node.
4. A leaf node is encoded as (`MPT_NODE_LEAF`, k, v), where k is a 2-tuple as above, and v is a pointer to a value.
5. A digest node is encoded as (`MPT_NODE_HASH`, d), where d is a Keccak256 digest.

On the other hand the values or payloads are represented differently depending on the particular trie.

4.1.1 State trie

The state trie payload contains the account data. Each account is stored in 4 contiguous memory addresses containing

1. the nonce,
2. the balance,

3. a pointer to the account's storage trie,
4. a hash of the account's code.

The storage trie payload in turn is a single word.

4.1.2 Transaction Trie

The transaction trie nodes contain the length of the RLP encoded transaction, followed by the bytes of the RLP encoding of the transaction.

4.1.3 Receipt Trie

The payload of the receipts trie is a receipt. Each receipt is stored as

1. the length in words of the payload,
2. the status,
3. the cumulative gas used,
4. the bloom filter, stored as 256 words.
5. the number of topics,
6. the topics
7. the data length,
8. the data.

4.2 Prover input format

The initial state of each trie is given by the prover as a nondeterministic input tape. This tape has a slightly different format:

1. An empty node is encoded as (MPT_NODE_EMPTY).
2. A branch node is encoded as (MPT_NODE_BRANCH, $v_?$, c_1, \dots, c_{16}). Here $v_?$ consists of a flag indicating whether a value is present, followed by the actual value payload if one is present. Each c_i is the encoding of a child node.

3. An extension node is encoded as $(\text{MPT_NODE_EXTENSION}, k, c)$, where k represents the part of the key associated with this extension, and is encoded as a 2-tuple $(\text{packed_nibbles}, \text{num_nibbles})$. c is a pointer to a child node.
4. A leaf node is encoded as $(\text{MPT_NODE_LEAF}, k, v)$, where k is a 2-tuple as above, and v is a value payload.
5. A digest node is encoded as $(\text{MPT_NODE_HASH}, d)$, where d is a Keccak256 digest.

Nodes are thus given in depth-first order, enabling natural recursive methods for encoding and decoding this format. The payload of state and receipt tries is given in the natural sequential way. The transaction and receipt payloads contain variable size data, thus the input is slightly different. The prover input for the transactions is the transaction RLP encoding preceded by its length. For the receipts is in the natural sequential way, except that topics and data are preceded by their lengths, respectively.

4.3 Encoding and Hashing

Encoding is done recursively starting from the trie root. Leaf, branch and extension nodes are encoded as the RLP encoding of list containing the hex prefix encoding of the node key as well as

Leaf Node: the encoding of the the payload,

Branch Node: the hash or encoding of the 16 children and the encoding of the payload,

Extension Node: the hash or encoding of the child and the encoding of the payload.

For the rest of the nodes we have:

Empty Node: the encoding of an empty node is $0x80$,

Digest Node: the encoding of a digest node stored as $(\text{MPT_HASH_NODE}, d)$ is d .

The payloads in turn are RLP encoded as follows

State Trie: Encoded as a list containing nonce, balance, storage trie hash and code hash.

Storage Trie: The RLP encoding of the value (thus the double RLP encoding)

Transaction Trie: The RLP encoded transaction.

Receipt Trie: Depending on the transaction type it's encoded as `RLP(RLP(receipt))` for Legacy transactions or `RLP(txn_type||RLP(receipt))` for transactions of type 1 or 2. Each receipt is encoded as a list containing:

1. the status,
2. the cumulative gas used,
3. the bloom filter, stored as a list of length 256.
4. the list of topics
5. the data string.

Once a node is encoded it is written to the `Segment::RlpRaw` segment as a sequence of bytes. Then the RLP encoded data is hashed if the length of the data is more than 32 bytes. Otherwise we return the encoding. Further details can be found in the [mpt hash folder](#).

5 CPU logic

The CPU is in charge of coordinating the different STARKs, proving the correct execution of the instructions it reads and guaranteeing that the final state of the EVM corresponds to the starting state after executing the input transactions. All design choices were made to make sure these properties can be adequately translated into constraints of degree at most 3 while minimizing the size of the different table traces (number of columns and number of rows).

In this section, we will detail some of these choices.

5.1 Kernel

The kernel is in charge of the proving logic. This section aims at providing a high level overview of this logic. For details about any specific part of the logic, one can consult the various “asm” files in the [“kernel” folder](#).

We prove a batch of transactions, split into segments. These proofs can later be aggregated recursively to prove a block. Proof aggregation is however not in the scope of this section. Here, we assume that we have an initial state

of the EVM, and we wish to prove that a batch of contiguous transactions was correctly executed, leading to a correct update of the state.

Since we process transactions and not entire blocks, a few intermediary values need to be provided by the prover. Indeed, to prove that the registers in the EVM state are correctly updated, we need to have access to their initial values. When aggregating proofs, we can also constrain those values to match from one batch to the next. Let us consider the example of the transaction number. Let n be the number of transactions executed so far in the current block. If the current proof is not a dummy one (we are indeed executing a batch of transactions), then the transaction number should be updated: $n := n + k$ with k the number of transactions in the batch. Otherwise, the number remains unchanged. We can easily constrain this update. When aggregating the previous transaction batch proof (lhs) with the current one (rhs), we also need to check that the output transaction number of lhs is the same as the input transaction number of rhs .

Those prover provided values are stored in memory prior to entering the kernel, and are used in the kernel to assert correct updates. The list of prover provided values necessary to the kernel is the following:

1. the number of the last transaction executed: t_n ,
2. the gas used before executing the current transactions: g_{-u_0} ,
3. the gas used after executing the current transactions: g_{-u_1} ,
4. the state, transaction and receipts MPTs before executing the current transactions: `tries0`,
5. the hash of all MPTs before executing the current transactions: `digests0`,
6. the hash of all MPTs after executing the current transactions: `digests1`,
7. the RLP encoding of the transactions.

Segment handling: An execution run is split into one or more segments. To ensure continuity, the first cycles of a segment are used to "load" segment data from the previous segment, and the last cycles to "save" segment data for the next segment. The number of CPU cycles of a segment is bounded by `MAX_CPU_CYCLES`, which can be tweaked for best performance. The segment data values are:

- the stack length,

- the stack top,
- the context,
- the `is_kernel` flag,
- the gas used,
- the program counter.

These values are stored as global metadata, and are loaded from (resp. written to) memory at the beginning (resp. at the end) of a segment. They are propagated between proofs as public values.

The initial memory of the first segment is fixed and contains:

- the kernel code,
- the shift table.

Initialization: The first step of a run consists in initializing:

- The initial transaction and receipt tries `tries0` are loaded from memory. The transaction and the receipt tries are hashed and the hashes are then compared to `digests_0`. For efficiency, the initial state trie will be hashed for verification at the end of the run.
- We load the transaction number $t.n$ and the current gas used $g.u_0$ from memory.

We start processing the transactions (if any) sequentially, provided in RLP encoded format.

The processing of the transaction returns a boolean “success” that indicates whether the transaction was executed successfully, along with the leftover gas.

The following step is then to update the receipts MPT. Here, we update the transaction’s bloom filter. We store “success”, the leftover gas, the transaction bloom filter and the logs in memory. We also store some additional information that facilitates the RLP encoding of the receipts later.

If there are any withdrawals, they are performed at this stage.

Finally, once the three MPTs have been updated, we need to carry out final checks:

- the gas used after the execution is equal to $g.u_1$,

- the new transaction number is $n + k$ with k the number of processed transactions,
- the initial state MPT is hashed and checked against `digests0`.
- the initial state MPT is updated to reflect the processed transactions, then the three final MPTs are hashed and checked against `digests1`.

Once those final checks are performed, the program halts.

5.2 Simple opcodes VS Syscalls

For simplicity and efficiency, EVM opcodes are categorized into two groups: “simple opcodes” and “syscalls”. Simple opcodes are generated directly in Rust, in [operation.rs](#). Every call to a simple opcode adds exactly one row to the [cpu table](#). Syscalls are more complex structures written with simple opcodes, in the kernel.

Whenever we encounter a syscall, we switch to kernel mode and execute its associated code. At the end of each syscall, we run `EXIT_KERNEL`, which resets the kernel mode to its state right before the syscall. It also sets the PC to point to the opcode right after the syscall.

Exceptions are handled differently for simple opcodes and syscalls. When necessary, simple opcodes throw an exception (see [5.7](#)). This activates the “exception flag” in the CPU and runs the exception operations. On the other hand, syscalls handle exceptions in the kernel directly.

5.3 Privileged instructions

To ease and speed-up proving time, the zkEVM supports custom, privileged instructions that can only be executed by the kernel. Any appearance of those privileged instructions in a contract bytecode for instance would result in an unprovable state.

In what follows, we denote by p_{BN} the characteristic of the BN254 curve base field, curve for which Ethereum supports the `ecAdd`, `ecMul` and `ecPairing` precompiles.

0x0C. `ADDFP254`. Pops 2 elements from the stack interpreted as BN254 base field elements, and pushes their addition modulo p_{BN} onto the stack.

0x0D. `MULFP254`. Pops 2 elements from the stack interpreted as BN254 base field elements, and pushes their product modulo p_{BN} onto the stack.

- 0x0E. **SUBFP254**. Pops 2 elements from the stack interpreted as BN254 base field elements, and pushes their difference modulo p_{BN} onto the stack. This instruction behaves similarly to the SUB (0x03) opcode, in that we subtract the second element of the stack from the initial (top) one.
- 0x0F. **SUBMOD**. Pops 3 elements from the stack, and pushes the modular difference of the first two elements of the stack by the third one. It is similar to the SUB instruction, with an extra pop for the custom modulus.
- 0x21. **KECCAK_GENERAL**. Pops 2 elements (a Memory address, followed by a length ℓ) and pushes the hash of the memory portion starting at the constructed address and of length ℓ . It is similar to KECCAK256 (0x20) instruction, but can be applied to any memory section (i.e. even privileged ones).
- 0x49. **PROVER_INPUT**. Pushes a single prover input onto the stack.
- 0xC0-0xDF. **MSTORE_32BYTES**. Pops 2 elements from the stack (a Memory address, and then a value), and pushes a new address' onto the stack. The value is being decomposed into bytes and written to memory, starting from the fetched address. The new address being pushed is computed as the initial address + the length of the byte sequence being written to memory. Note that similarly to PUSH (0x60-0x7F) instructions, there are 32 MSTORE_32BYTES instructions, each corresponding to a target byte length (length 0 is ignored, for the same reasons as MLOAD_32BYTES, see below). Writing to memory an integer fitting in n bytes with a length $\ell < n$ will result in the integer being truncated. On the other hand, specifying a length ℓ greater than the byte size of the value being written will result in padding with zeroes. This process is heavily used when resetting memory sections (by calling MSTORE_32BYTES.32 with the value 0).
- 0xF6. **GET_CONTEXT**. Pushes the current context onto the stack. The kernel always has context 0.
- 0xF7. **SET_CONTEXT**. Pops the top element of the stack and updates the current context to this value. It is usually used when calling another contract or precompile, to distinguish the caller from the callee.
- 0xF8. **MLOAD_32BYTES**. Pops 2 elements from the stack (a Memory address, and then a length ℓ), and pushes a value onto the stack. The pushed value corresponds to the U256 integer read from the big-endian sequence

of length ℓ from the memory address being fetched. Note that an empty length is not valid, nor is a length greater than 32 (as a U256 consists in at most 32 bytes). Missing these conditions will result in an unverifiable proof.

0xF9. **EXIT_KERNEL**. Pops 1 element from the stack. This instruction is used at the end of a syscall, before proceeding to the rest of the execution logic. The popped element, *kexit_info*, contains several pieces of information like the current program counter, the current amount of gas used, and whether we are in kernel (i.e. privileged) mode or not.

0xFB. **MLOAD_GENERAL**. Pops 1 elements (a Memory address), and pushes the value stored at this memory address onto the stack. It can read any memory location, general (similarly to MLOAD (0x51) instruction) or privileged.

0xFC. **MSTORE_GENERAL**. Pops 2 elements (a value and a Memory address), and writes the popped value from the stack at the fetched address. It can write to any memory location, general (similarly to MSTORE (0x52) / MSTORE8 (0x53) instructions) or privileged.

5.4 Memory addresses

Kernel operations deal with memory addresses as single U256 elements. However, when processing the operations to generate the proof witness, the CPU will decompose these into three components:

- context. The context of the memory address. The Kernel context is special, and has value 0.
- segment. The segment of the memory address, corresponding to a specific section given a context (eg. MPT data, global metadata, etc.).
- virtual. The offset of the memory address, within a segment given a context.

To easily retrieve these components, we scale them so that they can represent a memory address as:

$$\text{addr} = 2^{64} \cdot \text{context} + 2^{32} \cdot \text{segment} + \text{offset}$$

This allows to easily retrieve each component individually once a Memory address has been decomposed into 32-bit limbs.

5.5 Stack handling

5.5.1 Top of the stack

The majority of memory operations involve the stack. The stack is a segment in memory, and stack operations (popping or pushing) use the memory channels. Every CPU instruction performs between 0 and 3 pops, and may push at most once. However, for efficiency purposes, we hold the top of the stack in the first memory channel `current_row.mem_channels[0]`, only writing it in memory if necessary.

Motivation: See [this issue](#).

Top reading and writing: When a CPU instruction modifies the stack, it must update the top of the stack accordingly. There are three cases.

- **The instruction pops and pushes:** The new top of the stack is stored in `next_row.mem_channels[0]`; it may be computed by the instruction, or it could be read from memory. In either case, the instruction is responsible for setting `next_row.mem_channels[0]`'s flags and address columns correctly. After use, the previous top of the stack is discarded and doesn't need to be written in memory.
- **The instruction pushes, but doesn't pop:** The new top of the stack is stored in `next_row.mem_channels[0]`; it may be computed by the instruction, or it could be read from memory. In either case, the instruction is responsible for setting `next_row.mem_channels[0]`'s flags and address columns correctly. If the stack wasn't empty (`current_row.stack_len > 0`), the instruction performs a memory read in `current_row.partial_channel`. `current_row.partial_channel` shares its values with `current_row.mem_channels[0]` (which holds the current top of the stack). If the stack was empty, `current_row.partial_channel` is disabled.
- **The instruction pops, but doesn't push:** After use, the current top of the stack is discarded and doesn't need to be written in memory. If the stack isn't empty now (`current_row.stack_len > num_pops`), the new top of the stack is set in `next_row.mem_channels[0]` with a memory read from the stack segment. If the stack is now empty, `next_row.mem_channels[0]` is disabled.

In the last two cases, there is an edge case if `current_row.stack_len` is equal to a `special_len`. For a strictly pushing instruction, this happens if the

stack is empty, and `special_len = 0`. For a strictly popping instruction, this happens if the next stack is empty, i.e. if all remaining elements are popped, and `special_len = num_pops`. Note that we do not need to check for values below `num_pops`, since this would be a stack underflow exception which is handled separately. The edge case is detected with the compound flag

$$1 - \text{not_special_len} * \text{stack_inv_aux},$$

where

$$\text{not_special_len} = \text{current_row} - \text{special_len}$$

and `stack_inv_aux` is constrained to be the modular inverse of `not_special_len` if it's non-zero, or 0 otherwise. The flag is 1 if `stack_len` is equal to `special_len`, and 0 otherwise.

This logic can be found in code in the `eval_packed_one` function of [stack.rs](#). The function multiplies all of the stack constraints with the degree 1 filter associated with the current instruction.

Operation flag merging: To reduce the total number of columns, many operation flags are merged together (e.g. `DUP` and `SWAP`) and are distinguished with the binary decomposition of their opcodes. The filter for a merged operation is now of degree 2: for example, `is_swap = dup_swap * opcode_bits[4]` since the 4th bit is set to 1 for a `SWAP` and 0 for a `DUP`. If the two instructions have different stack behaviors, this can be a problem: `eval_packed_one`'s constraints are already of degree 3 and it can't support degree 2 filters.

When this happens, stack constraints are defined manually in the operation's dedicated file (e.g. `dup_swap.rs`). Implementation details vary case-by-case and can be found in the files.

5.5.2 Stack length checking

The CPU must make sure that the stack length never goes below zero and, in user mode, never grows beyond the maximum stack size. When this happens, an honest prover should trigger the corresponding exception. If a malicious prover doesn't trigger the exception, constraints must fail the proof.

Stack underflow: There is no explicit constraint checking for stack underflow. An underflow happens when the CPU tries to pop the empty stack, which would perform a memory read at virtual address `-1`. Such a read cannot succeed: in Memory, the range-check argument requires the gap between two consecutive addresses to be lower than the length of the Memory trace.

Since the prime of the Plonky2 field is 64-bit long, this would require a Memory trace longer than 2^{32} .

Stack overflow: An instruction can only push at most once, meaning that an overflow occurs whenever the stack length is exactly one more than the maximum stack size ($1024 + 1$) in user mode. To constrain this, the column `stack_len_bounds_aux` contains:

- the modular inverse of `stack_len - 1025` if we’re in user mode and `stack_len \neq 1025`,
- 0 if `stack_len = 1025` or if we’re in kernel mode.

Then overflow can be checked with the flag

$$(1 - \text{is_kernel_mode}) - \text{stack_len} * \text{stack_len_bounds_aux}.$$

The flag is 1 if `stack_len = 1025` and we’re in user mode, and 0 otherwise.

Because `stack_len_bounds_aux` is a shared general column, we only check this constraint after an instruction that can actually trigger an overflow, i.e. a pushing, non-popping instruction.

5.6 Gas handling

5.6.1 Out of gas errors

The CPU table has a “gas” register that keeps track of the gas used by the transaction so far.

The crucial invariant in our out-of-gas checking method is that at any point in the program’s execution, we have not used more gas than we have available; that is “gas” is at most the gas allocation for the transaction (which is stored separately by the kernel). We assume that the gas allocation will never be 2^{32} or more, so if “gas” does not fit in one limb, then we’ve run out of gas.

When a native instruction (one that is not a syscall) is executed, a constraint ensures that the “gas” register is increased by the correct amount. This is not automatic for syscalls; the syscall handler itself must calculate and charge the appropriate amount.

If everything goes smoothly and we have not run out of gas, “gas” should be no more than the gas allowance at the point that we STOP, REVERT, stack overflow, or whatever. Indeed, because we assume that the gas overflow handler is invoked *as soon as* we’ve run out of gas, all these termination methods verify that `gas \leq allowance`, and jump to `exc_out_of_gas` if this is not the case. This is also true for the out-of-gas handler, which checks that:

1. we have not yet run out of gas
2. we are about to run out of gas

and “PANIC” if either of those statements does not hold.

When we do run out of gas, however, this event must be handled. Syscalls are responsible for checking that their execution would not cause the transaction to run out of gas. If the syscall detects that it would need to charge more gas than available, it aborts the transaction (or the current code) by jumping to `fault_exception`. In fact, `fault_exception` is in charge of handling all exceptional halts in the kernel.

Native instructions do this differently. If the prover notices that execution of the instruction would cause an out-of-gas error, it must jump to the appropriate handler instead of executing the instruction. (The handler contains special code that PANICs if the prover invoked it incorrectly.)

5.6.2 Overflow

We must be careful to ensure that “gas” does not overflow to prevent denial of service attacks.

Note that a syscall cannot be the instruction that causes an overflow. This is because every syscall is required to verify that its execution does not cause us to exceed the gas limit. Upon entry into a syscall, a constraint verifies that $\text{gas} < 2^{32}$. Some syscalls may have to be careful to ensure that the gas check is performed correctly (for example, that overflow modulo 2^{256} does not occur). So we can assume that upon entry and exit out of a syscall, $\text{gas} < 2^{32}$.

Similarly, native instructions alone cannot cause wraparound. The most expensive instruction, JUMPI, costs 10 gas. Even if we were to execute 2^{32} consecutive JUMPI instructions, the maximum length of a trace, we are nowhere close to consuming $2^{64} - 2^{32} + 1$ (= Goldilocks prime) gas.

The final scenario we must tackle is an expensive syscall followed by many expensive native instructions. Upon exit from a syscall, $\text{gas} < 2^{32}$. Again, even if that syscall is followed by 2^{32} native instructions of cost 10, we do not see wraparound modulo Goldilocks.

5.7 Exceptions

Sometimes, when executing user code (i.e. contract or transaction code), the EVM halts exceptionally (i.e. outside of a STOP, a RETURN or a REVERT). When this happens, the CPU table invokes a special instruction with a dedicated operation flag `exception`. Exceptions can only happen in user mode;

triggering an exception in kernel mode would make the proof unverifiable. No matter the exception, the handling is the same:

- The opcode which would trigger the exception is not executed. The operation flag set is `exception` instead of the opcode's flag.
- We push a value to the stack which contains: the current program counter (to retrieve the faulty opcode), and the current value of `gas_used`. The program counter is then set to the corresponding exception handler in the kernel (e.g. `exc_out_of_gas`).
- The exception handler verifies that the given exception would indeed be triggered by the faulty opcode. If this is not the case (if the exception has already happened or if it doesn't happen after executing the faulty opcode), then the kernel panics: there was an issue during witness generation.
- The kernel consumes the remaining gas and returns from the current context with `success` set to 0 to indicate an execution failure.

Here is the list of the possible exceptions:

Out of gas: Raised when a native instruction (i.e. not a syscall) in user mode pushes the amount of gas used over the current gas limit. When this happens, the EVM jumps to `exc_out_of_gas`. The kernel then checks that the consumed gas is currently below the gas limit, and that adding the gas cost of the faulty instruction pushes it over it. If the exception is not raised, the prover will panic when returning from the execution: the remaining gas is checked to be positive after STOP, RETURN or REVERT.

Invalid opcode: Raised when the read opcode is invalid. It means either that it doesn't exist, or that it's a privileged instruction and thus not available in user mode. When this happens, the EVM jumps to `exc_invalid_opcode`. The kernel then checks that the given opcode is indeed invalid. If the exception is not raised, decoding constraints ensure no operation flag is set to 1, which would make it a padding row. Halting constraints would then make the proof unverifiable.

Stack underflow: Raised when an instruction which pops from the stack is called when the stack doesn't have enough elements. When this happens, the EVM jumps to `exc_stack_overflow`. The kernel then checks that the current stack length is smaller than the minimum stack length required by the faulty opcode. If the exception is not raised, the popping memory operation's address offset would underflow, and the Memory range check would require the Memory trace to be too large ($> 2^{32}$).

Invalid JUMP destination: Raised when the program counter jumps to an invalid location (i.e. not a JUMPDEST). When this happens, the EVM jumps to `exc_invalid_jump_destination`. The kernel then checks that the opcode is a JUMP, and that the destination is not a JUMPDEST by checking the JUMPDEST segment. If the exception is not raised, jumping constraints will fail the proof.

Invalid JUMPI destination: Same as the above, for JUMPI.

Stack overflow: Raised when a pushing instruction in user mode pushes the stack over 1024. When this happens, the EVM jumps to `exc_stack_overflow`. The kernel then checks that the current stack length is exactly equal to 1024 (since an instruction can only push once at most), and that the faulty instruction is pushing. If the exception is not raised, stack constraints ensure that a stack length of 1025 in user mode will fail the proof.

5.8 Linked lists

Individual account information are contained in the state and the storage MPTs. However, accessing and modifying MPT data requires heavy trie traversal, insertion and deletion functions. To alleviate these costs, during an execution run, we store all account information in linked list structures and only modify the state trie at the end of the run.

Our linked list construction guarantees these properties:

- A linked list is cyclic. The last element's successor is the first element.
- A linked list is always sorted by a certain index, which can be one or more fields of an element.
- The last element of a linked list is MAX, whose index is always higher than any possible index value.
- An index cannot appear twice in the linked list.

These properties allows us to efficiently modify the list.

Search To search a node given its index, we provide via `PROVER_INPUT` a pointer to its predecessor p . We first check that p 's index is strictly lower than the node index, if not, the provided pointer is invalid. Then, we check s , p 's successor. If s 's index is equal to the node index, we found the node. If s 's

index is lower than the node index, then the provided p was invalid. If s 's index is greater than the node index, then the node doesn't exist.

Insertion To insert a node given its index, we provide via `PROVER_INPUT` a pointer to its predecessor p . We first check that p 's index is strictly lower than the node index, if not, the provided pointer is invalid. Then, we check s , p 's successor, and make sure that s is strictly greater than the node index. We create a new node, and make it p 's successor; then we make s the new node's successor.

Deletion To delete a node given its index, we provide via `PROVER_INPUT` a pointer to its predecessor p . We check that p 's successor is equal to the node index; if not either p is invalid or the node doesn't exist. Then we set p 's successor to the node's successor. To indicate that the node is now deleted and to make sure that it's never accessed again, we set its next pointer to `MAX`.

We maintain two linked lists: one for the state accounts and one for the storage slots.

Account linked list

An account node is made of four memory cells:

- The account key (the hash of the account address). This is the index of the node.
- A pointer to the account payload, in segment `@TrieData`.
- A pointer to the initial account payload, in segment `@TrieData`. This is the value of the account at the beginning of the execution, before processing any transaction. This payload never changes.
- A pointer to the next node (which points to the next node's account key).

Storage linked list

A storage node is made of five memory cells:

- The account key (the hash of the account address).
- The slot key (the hash of the slot). Nodes are indexed by (`account_key`, `slot_key`).

- The slot value.
- The initial slot value. This is the value of the account at the beginning of the execution, before processing any transaction. It never changes.
- A pointer to the next node (which points to the next node's account key).

References

- [1] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Scalable, transparent, and post-quantum secure computational integrity.” Cryptology ePrint Archive, Report 2018/046, 2018. <https://ia.cr/2018/046>.
- [2] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.