

# The Polygon Zero zkEVM

DRAFT  
October 14, 2022

## **Abstract**

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

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>STARK framework</b>	<b>3</b>
2.1	Cost model . . . . .	3
2.2	Field selection . . . . .	3
2.3	Cross-table lookups . . . . .	4
<b>3</b>	<b>Tables</b>	<b>4</b>
3.1	CPU . . . . .	4
3.2	Arithmetic . . . . .	4
3.3	Logic . . . . .	4
3.4	Memory . . . . .	4
3.4.1	Virtual memory . . . . .	5
3.4.2	Timestamps . . . . .	6
3.5	Keccak-f . . . . .	6
3.6	Keccak sponge . . . . .	6
<b>4</b>	<b>Merkle Patricia tries</b>	<b>6</b>
4.1	Internal memory format . . . . .	6
4.2	Prover input format . . . . .	7
<b>5</b>	<b>Privileged instructions</b>	<b>7</b>

# 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

TODO

# 3 Tables

## 3.1 CPU

TODO

## 3.2 Arithmetic

TODO

## 3.3 Logic

TODO

## 3.4 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.  $\tau$ , the timestamp of the operation

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

1. Verify the ordering by checking that  $(a_i, \tau_i) < (a_{i+1}, \tau_{i+1})$  for each consecutive pair.

2. Enumerate the purportedly-ordered log while tracking a “current” value  $c$ , which is initially zero.<sup>1</sup>
  - (a) Upon observing an address which doesn’t match that of the previous row, set  $c \leftarrow 0$ .
  - (b) Upon observing a write, set  $c \leftarrow v$ .
  - (c) Upon observing a read, check that  $v = c$ .

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 are equal. An honest prover will set

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

We then impose the following transition constraints:

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

The last 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.

Finally, the iterative checks can be arithmetized by introducing a trace column for the current value  $c$ . We add a boundary constraint  $c_0 = 0$ , and the following transition constraints:

1.  $v_{\text{from},i} = c_i$ ,
2.  $c_{i+1} = e_i v_{\text{to},i}$ .

This is out of date, we don't actually need a  $c$  column.

### 3.4.1 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,

---

<sup>1</sup>EVM memory is zero-initialized.

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.

### 3.4.2 Timestamps

TODO: Explain  $\tau = \text{NUM\_CHANNELS} \times \text{cycle} + \text{channel}$ .

### 3.5 Keccak-f

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

### 3.6 Keccak sponge

This table computes the Keccak256 hash, a sponge-based hash built on top of the Keccak-f[1600] permutation.

## 4 Merkle Patricia tries

### 4.1 Internal memory format

Without our zkEVM's kernel memory,

1. An empty node is encoded as `(MPT_NODE_EMPTY)`.
2. A branch node is encoded as `(MPT_NODE_BRANCH, c1, ..., c16, 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.

## 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 (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 value payload.
5. A digest node is encoded as (MPT\_NODE\_HASH,  $d$ ), where  $d$  is a Keccak256 digest.

In the current implementation, we use a length prefix rather than a is-present prefix, but we plan to change that.

Nodes are thus given in depth-first order, enabling natural recursive methods for encoding and decoding this format.

## 5 Privileged instructions

0xFB. MLOAD\_GENERAL. Returns

0xFC. MSTORE\_GENERAL. Returns

TODO. STACK\_SIZE. Returns

## 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>.