The zkEVM Architecture
Part VII: Feijoa (And EIP-4844)

This is Work In Progress

Polygon zkEVM & Universitat Politècnica de Catalunya (UPC)

Marc Guzman-Albiol <marc.guzman.albiol@upc.edu>
Jose Luis Muñoz-Tapia <jose.luis.munoz@upc.edu>

March 28, 2024

Ethereum Data Sharding

Feijoa

Integrity of execution is not all a rollup needs.

A rollup needs to guarantee one or both of the following:

a) Its full-state is recoverable.

b) Additionally, that one or more concrete transactions were executed correctly.

To achieve the first property we can relay on **availability of state diffs**.

To achieve both, the L2 **transaction data has to be available**.

- Data availability options:
    a) Only commitments are available on chain (validium).
    b) State differences are available on chain (state diff rollup).
    c) The complete L2 transactions are available on chain (full rollup).

- Availability is provided in the **calldata** area of L1 transactions.

- Regarding **calldata**:
    - Is accessible by the EVM execution.
    - Is relatively expensive (16 gas per non-zero byte and 4 gas per zero byte).

Ethereum Data Sharding: Danksharding

- Data sharding is the way in which is planned that Ethereum becomes a truly scalable blockchain.

- The aim is to make transactions on Layers 2 as cheap as possible for users and should scale Ethereum to >100,000 transactions per second.

- **Danksharding**[1] is the concrete data sharding scheme used by Ethereum.

- The main ideas behind Danksharding are:

  1. Allow a new type of transaction that is able to carry one or more **blobs**, which stands for "Binary Large OBject".
  2. A blob stores binary data, typically large in size and **unstructured** from the point of view of the transporting layer, in this case, blobs have **no meaning for the L1 execution layer** but they are managed by a layer 2.
  3. The idea to make the network scalable is that blobs do not persist forever in the network nodes but they have a limited lifetime (**data pruning**), as a result, blobs should be much **cheaper** than calldata.

---

[1]The name Danksharding is due to the researcher Dankrad Feist.

APPLICATION

EXECUTION

commit to
blobdata

CONSENSUS

blobdata

COMMUNICATION
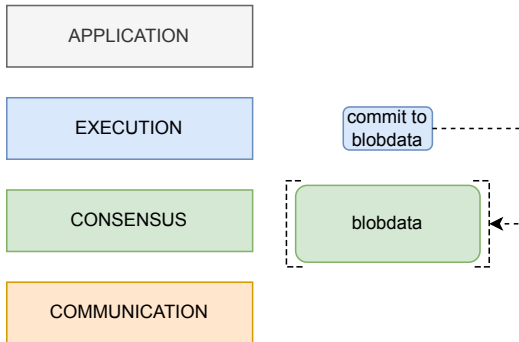
- Storage of blob carrying transactions (blobdata) is only visible at the consensus client NOT at the execution client.
- The execution layer does not have direct access to blob transactions only to commitments to blobs.
- Commitment goes on chain forever (but is small).
- Data stays for a while (weeks) and then disapears.

Uses the curve BLS12-381 which is pairing friendly.

The commitment to each blob is in the format of a versioned hash. It is 32 bytes value where the first byte is the version, currently set to 0x010x01, followed by the last 31 bytes of SHA256 hash of the KZG commitment of the blob, i.e. version_byte + SHA256(KZG(blob))[1:]. The rationale is to keep the possibility of changing the commitment scheme from KZG to something else without breaking the format, just in case KZG commitments are deemed to be not as safe as desired in the future, for instance if quantum computers become practical. The field blob_versioned_hashes denotes a list of commitments to the blobs included in the transaction. Notice that a blob-carrying transaction can carry more than one blob.

Opcode BLOBHASH

It is used as BLOBHASH index Where index is the position of the blob in the blob carrying transaction

Blob Carrying Transaction

In the blob transaction type:

- Data is left outside the transaction itself.
- This is now a responsability of the consensus layer.
- The transaction contains a committment to the blob data but not the data itself.
- Same functionality as EIP-1559 transaction including calldata but not RLP-encoded.
- The transaction is not encoded with RLP, so that it Merklelizes nicely, which is suitable for layers 2.
- The blob Carrying Transaction uses two extra fields: data_fee and data_hashes.

incorporate

Indep of amount of data but double the data to send.

- We do not use merkle trees for creating the commitment because it breaks the structure of polynomials (we need to use FRI or similars to do this). - We use a commitment that always commits to a polynomial

· When creating the L2 Tx bundle the rollup operator can do compression and so on.

- New transaction type that contains commitments to blobs.
- Opcode that outputs the i'th blob versioned hash in the current transaction.
- Blob verification precompile.
- Point evaluation precompile.

Easy proof of equivalence between multiple polynomial commitment schemes to the same data

Suppose you have multiple polynomial commitments $C_1, \ldots C_k$, under k different commitment schemes (eg. Kate, FRI, something bulletproof-based, DARK, etc), and you want to prove that they all commit to the same polynomial P.

We can prove this easily:

Let $z = \text{hash}(C_1 \ldots C_k)$, where we interpret z as an evaluation point at which P can be evaluated.

danksharding: merged fee market for all the shards

Danksharding and Proto-Danksharding

- **Proto-Danksharding** (note that the name includes Proto, which comes from Protolambda, another researcher) is an intermediate step along the way to achieve "full Danksharding".
- Proto-Danksharding is also known as EIP-4844 (the document in which it is defined).

3 gas per byte

1.4 MB

number of blobs per block

goal 10/100 times less gas.
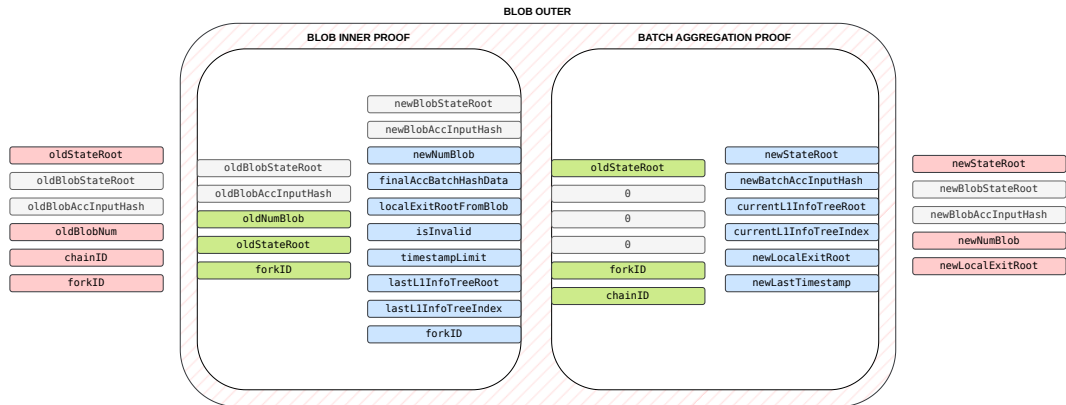
sepolia.blobscan.com

Add figures of video Eth Denver

Outline

.

blobOuter Circuit

- This circuit is in charge of aggregating a proof π_{re2} (or π_{re1} in the edge case of "aggregating" a single batch) and a proof $\pi_{blobInner}$ generated using the execution trace generated by the blob-specific ROM which is in charge of validating the blob data.

- Moreover, its responsibility extends to conducting integrity checks on the data received from both sources, ensuring the coherence between the blob data and the batch data, which should be the same, except in some scenarios that will be specified later on.

BLOB OUTER

BLOB INNER PROOF

BATCH AGGREGATION PROOF

oldStateRoot

oldBlobStateRoot

oldBlobAccInputHash

oldBlobNum

chainID

forkID

oldBlobStateRoot

oldBlobAccInputHash

oldNumBlob

oldStateRoot

forkID

newBlobStateRoot

newBlobAccInputHash

newNumBlob

finalAccBatchHashData

localExitRootFromBlob

isInvalid

timestampLimit

lastL1InfoTreeRoot

lastL1InfoTreeIndex

forkID

oldStateRoot

0

0

0

forkID

chainID

newStateRoot

newBatchAccInputHash

currentL1InfoTreeRoot

currentL1InfoTreeIndex

newLocalExitRoot

newLastTimestamp

newStateRoot

newBlobStateRoot

newBlobAccInputHash

newNumBlob

newLocalExitRoot

isValidBlob Signal

- First of all, the circuit creates a signal named **isValidBlob** that asserts whether the blob processing carried on by the **blobInner** processing is deemed valid. This boolean signal is 1 if and only if this two conditions are satisfied:
  - At least one element within the 8-element array **finalAccBatchData** is non-zero.
  - The input signal **isInvalid** produced by the **blobInner** ROM is 0.
- Whenever the signal **isValidBlob** signal is 0, the batch verifier is automatically disabled as the transmitted blob batches would not align with the provided batch data in the aggregated proof.
- In such instances, the prover can input any signals he/she wish, but we must finally assert a proof of no state change.
- Thus, this signal it is used extensively in the circuit because there are constraints that do not make sense if the blob data processing has resulted invalid.

Other Sources of Failure

- However, errors in the input data may arise from scenarios other than the validity of the blob parsing. Specifically, there are two scenarios to consider:
  - When the last timestamp recorded on the batch newLastTimestamp is greater than the timestamp limit specified in the blob timestampLimit $\rightarrow$ isValidTimestamp.
  - When the selected information tree index is incorrect $\rightarrow$ isValidL1InfoTreeIndex
- The three error controlling signals will be accumulated in isValid signal, which will only be 1 if and only if each of them is 1. That is

$$\text{isValid} = 1 \iff \begin{array}{l} \text{isValidBlob} = 1 \text{ and} \\ \text{isValidTimestamp} = 1 \text{ and} \\ \text{isValidL1InfoTreeIndex} = 1. \end{array}$$
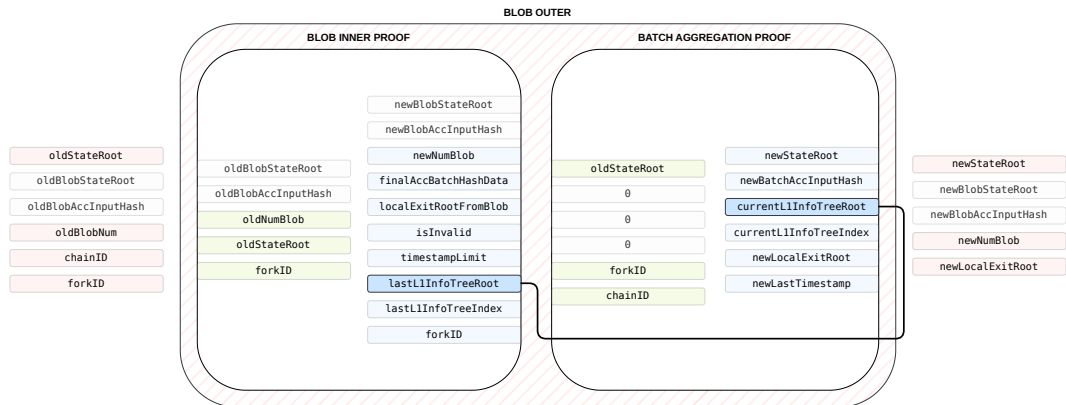
- Component-wise assertion that `finalAccBatchHashData`, computed in `blobInner`, coincides with `newBatchAccInputHash`, computed within the batch aggregation.
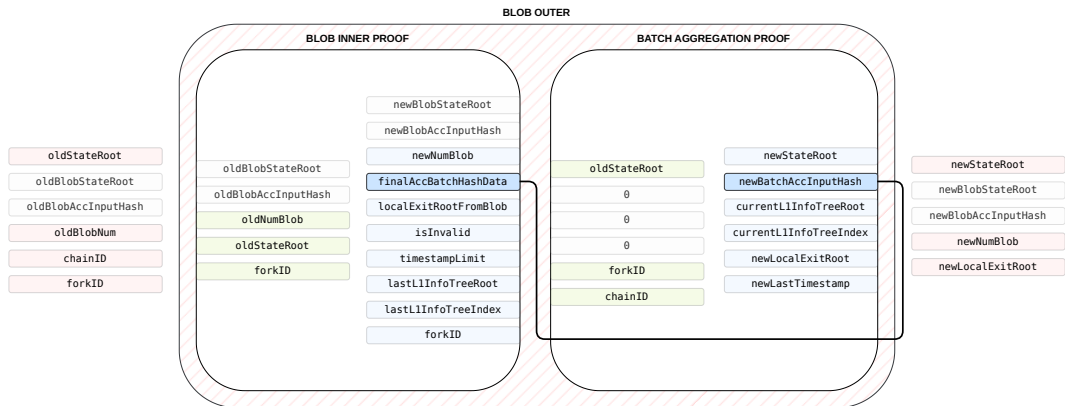
- Component-wise assertion that the root of the L1 Info Tree from the blob data
  lastL1InfoTreeRoot coincides with the one coming with the batch data
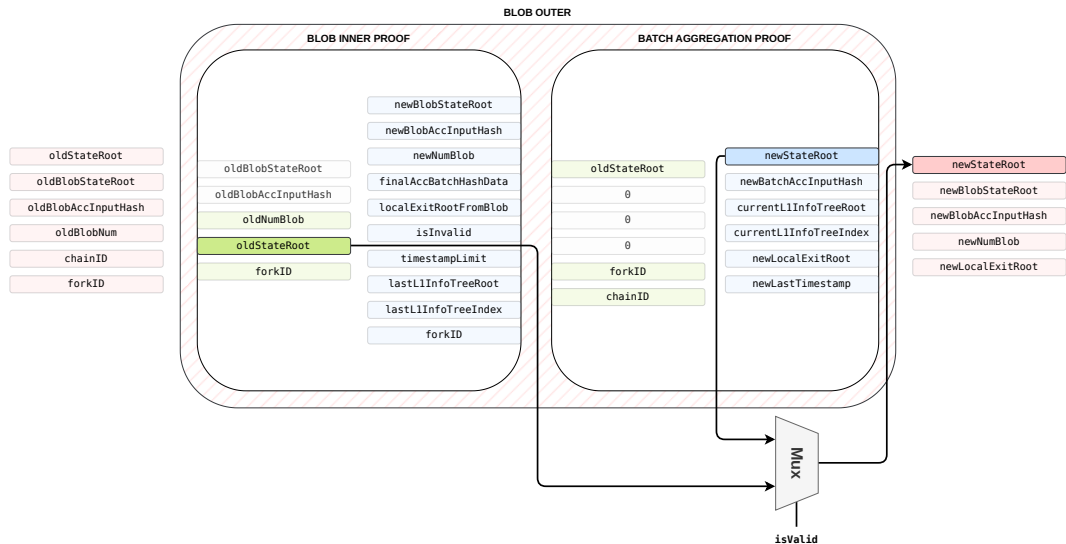 currentL1InfoTreeRoot.

- Component-wise assertion that the old state root obtained from the blob data matches the old state root retrieved from the batch aggregation.
- Furthermore, the output of the old state root from the blob outer circuit is assigned to be one of these roots. This check is performed in all scenarios.
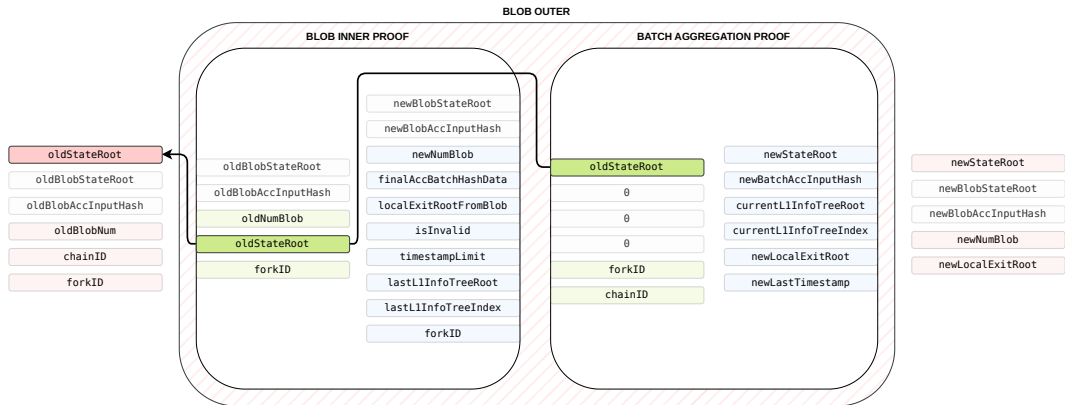
## New State Root Integrity Check and Output Assignment

- The circuit's output signal **newStateRoot** is properly determined using a multiplexer.
- When the **isValid** signal is 0 the state root cannot be updated due to some errors.
- In such scenarios, it should be assigned the value of **oldStateRoot** obtained from the **blobInner** circuit.
- It's important to note that we cannot utilize the state root from the batch aggregation because, if the verifier is disabled by the **isValidBlob** signal, the prover has the freedom to provide arbitrary inputs.
- Conversely, when the **isValid** signal is 1, indicating correctness, we can simply assign the **newStateRoot** obtained from the batch aggregation.

- The circuit's output signal newLocalExitRoot is properly determined using a multiplexer.
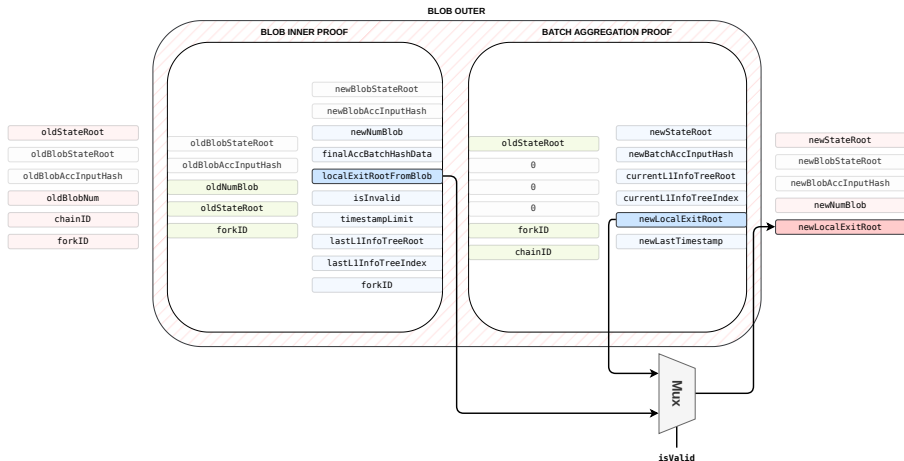
Output Blob Numbers Assignment

- The circuit's output signals **oldBlobNum** and **newBlobNum** are properly assigned from the data coming from the **blobInner** circuit.