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>

Version: 792c01a24110b5d15f1f271aa33d27e6983956ee

April 11, 2024

Outline

Why Data Availability?

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 Pre-EIP 4844

- 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).

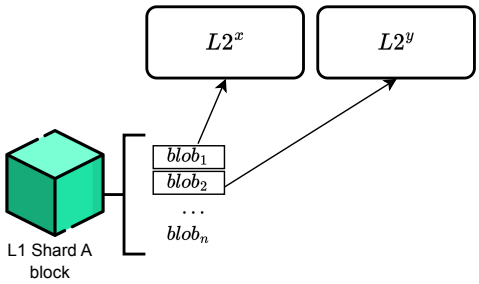
Ethereum Data Sharding: Danksharding

- 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.

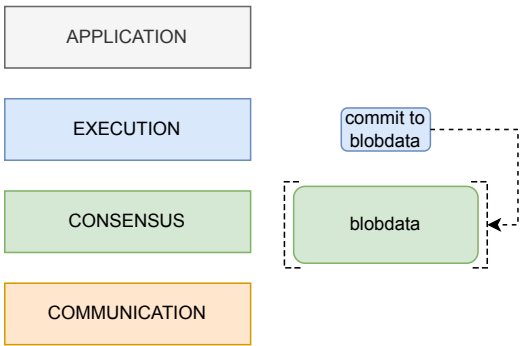
[1] The name Danksharding is due to the researcher Dankrad Feist.

blobs

Instead of transactions, shard blocks contain **blobs** (binary large objects) that designed to be interpreted by a top layer.



Danksharding and Blockchain Layers



- 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.

Commit to BlobData: KZG

Uses the curve BLS12-381 which is pairing friendly.

Commit to BlobData: Versioned Hash

- The commitment to each blob is in the format of a versioned hash.
- The versiones hash is a 32 bytes value:

$$\texttt{version\_byte} + \texttt{SHA256(KZG(blob))[1:]}$$

  - The first byte is the version, currently set to `0x01`.
  - The following 31 bytes are the last 31 bytes of SHA256 hash of the KZG commitment of the blob, i.e:
- The rationale to use versioned hashes and not directly the KZG blob commitment is to keep the **possibility of changing the commitment scheme** from KZG to something else without breaking the format.
- E.g. in case KZG commitments are deemed to be not as safe as desired in the future, for instance if quantum computers become practical.

Blob Carrying Transaction i

In the blob transaction type:

- In the transaction, 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.
- 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.

Blob Carrying Transaction ii

- TransactionPayload is the RLP serialization of the following TransactionPayloadBody: [chain_id, nonce, max_priority_fee_per_gas, max_fee_per_gas, gas_limit, to, value, data, access_list, max_fee_per_blob_gas, blob_versioned_hashes, y_parity, r, s]
- The fields chain_id, nonce, max_priority_fee_per_gas, max_fee_per_gas, gas_limit, value, data, and access_list follow the same semantics as EIP-1559.
- The field to deviates slightly from the semantics with the exception that it MUST NOT be nil and therefore must always represent a 20-byte address. This means that blob transactions cannot have the form of a create transaction.
- The field max_fee_per_blob_gas is a uint256 and the field blob_versioned_hashes represents a list of hash outputs from kzg_to_versioned_hash.
- The EIP-2718 ReceiptPayload for this transaction is rlp([status, cumulative_transaction_gas_used, logs_bloom, logs]).

Blob Carrying Transaction iii

- Signature
- The signature values y_parity, r, and s are calculated by constructing a secp256k1 signature over the following digest:
  keccak256(BLOB_TX_TYPE || rlp([chain_id, nonce, max_priority_fee_per_gas, max_fee_per_gas, gas_limit, to, value, data, access_list, max_fee_per_blob_gas, blob_versioned_hashes])).

Opcode BLOBHASH

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

Block Header extension

The current header encoding is extended with two new 64-bit unsigned integer fields:

- blob_gas_used is the total amount of blob gas consumed by the transactions within the block.
- excess_blob_gas is a running total of blob gas consumed in excess of the target, prior to the block. Blocks with above-target blob gas consumption increase this value, blocks with below-target blob gas consumption decrease it (bounded at 0).

Proposer-Builder Separation (PBS) and Danksharding

Note. There is no finalized specification yet.

Rationale:

To generate blocks with blobs, someone will have to compute proofs for up to 64 MB of blob data in less than 1 second. This will probably require specialized builders that can dedicate fairly substantial hardware to the task. However, in the current situation block building could become increasingly centralized around more sophisticated and powerful operators anyway due to MEV extraction. Proposer-builder separation is a way to embrace this reality and prevent it from exerting centralizing force on block validation (the important part) or the distribution of staking rewards. A great side-benefit is that the specialized block builders are also willing and able to compute the necessary data proofs for Danksharding.

https://ethereum.org/ca/roadmap/pbs

Data Availability (DA) sampling

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

Proof of Equivalence

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

Suppose you have multiple polynomial commitments C1, ... Ck, 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=hash(C1....Ck), where we interpret z as an evaluation point at which P can be evaluated.

Blobdata Market

danksharding: merged fee market for all the shards

Danksharding and Proto-Danksharding i

- **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).

Changes in the execution-layer

- 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.

Blob Verification Precompile

Point Evaluation Precompile

The Barycentric Formula: Motivation i

- Recall that the embedded field of the EC BLS12381 is of order $r = \texttt{0x73...01} \approx 2^{255}$.
- Consider the subgroup $\langle\omega\rangle \subset \mathbb{F}_r^*$ where ω is a primitive d-th root of unity, being $d = 4096$.
- That is, $\langle\omega\rangle$ is the group of the group of d-th roots of unity

$$\{\omega^0 = 1, \omega, \omega^2, \ldots, \omega^{d-1}\}.$$

- Recall that we want to evaluate a polynomial $f(X) \in \mathbb{F}_r[X]_{<\deg d}$ in a point $z \in \mathbb{F}_r$ where we are only given the evaluations $f_0, f_1, \ldots, f_{d-1}$ of f at the elements of $\langle\omega\rangle$

$$f(\omega^i) = f_i, \quad i \in \{0, 1, \ldots, d-1\}.$$

The Barycentric Formula: Motivation ii

- **Naive idea**: Represent the polynomial $f(X)$ in the Lagrange basis

$$f(X) = \sum_{i=0}^{d-1} f_i L_i(X),$$

being $L_i(X)$ the Lagrange polynomials:

$$L_i(\omega^j) = \begin{cases} 1, & i = j, \\ 0, & \text{otherwise} \end{cases}$$

The Barycentric Formula: Motivation iii

- Now, evaluate all $L_i(X)$ at z and compute $f(z)$

$$f(z) = \sum_{i=0}^{d-1} f_i L_i(z).$$

- **Problem:** The Lagrange polynomials have degree $d-1$ so, evaluating all of them at z using Horner's rule (which gives us $\mathcal{O}(d)$ for each polynomial) will mean a time complexity of $\mathcal{O}(d^2)$, which is quite slow.
- We can improve this to $\mathcal{O}(d)$ as we will see in next slides.

The Barycentric Formula: Derivation i

- Observe that the polynomials $L_i(X)$ can be obtained as follows:

$$L_i(X) = \frac{(X-\omega^0)(X-\omega^1)(X-\omega^2)\cdots(X-\omega^{i-1})(X-\omega^{i+1})\cdots(X-\omega^{d-1})}{(\omega^i-\omega^0)(\omega^i-\omega^1)(\omega^i-\omega^2)\cdots(\omega^i-\omega^{i-1})(\omega^i-\omega^{i+1})\cdots(\omega^i-\omega^{d-1})}.$$

- If we denote $H(X) = (X-\omega^0)(X-\omega^1)(X-\omega^2)\cdots(x-\omega^{d-1})$, note that

$$H'(\omega^i) = \prod_{i\neq j}(\omega^i-\omega^j)$$

  because all the other therms have the factor $(X-\omega^i)$, which vanishes when $X=\omega^i$.

- Henceforth, we can write the Lagrange polynomials as follows:

$$L_i(X) = \frac{H(X)}{H'(\omega^i)(X-\omega^i)}.$$

The Barycentric Formula: Derivation ii

- It works:

$$L_i(\omega^j) = \frac{H(\omega^j)}{H'(\omega^i)(\omega^j - \omega^i)} = \frac{(\omega^j - \omega^0)(\omega^j - \omega^1)\cdots\cancel{(\omega^j - \omega^j)}\cdots(\omega^j - \omega^{d-1})}{\cancel{(\omega^j - \omega^j)}\cdot\prod_{i\neq j}(\omega^j - \omega^i)} = 1$$

$$L_i(\omega^j) = \frac{H(\omega^j)}{H'(\omega^j)(\omega^j - \omega^i)} = 0,$$

  when $i \neq j$ because $H(\omega^j) = 0$ and the denominator does not vanishes.

- So we can express $f(X)$ as follows:

$$f(X) = \sum_{i=0}^{d-1} f_i L_i(X) = H(X)\sum_{i=0}^{d-1}\frac{f_i}{H'(\omega^i)}\frac{1}{X - \omega^i}$$

- If we pre-compute the quotients $1/H'(\omega^j)$ for all j, evaluating $f(X)$ using the formula above requires $\mathcal{O}(d)$.

The Barycentric Formula: Roots of Unity i

- We can further improve the later formula by using the fact that our interpolation domain are roots of unity.
- First of all, note that, since $X^d - 1$ is a degree d polynomial that vanishes in all ω^i, we should have

$$H(X) = X^d - 1.$$

- Recall that

$$X^d - 1 = (X - 1)(X^{d-1} + X^{d-2} + \cdots + X + 1).$$

- Observe that

$$L_0(X) = \frac{X^d - 1}{K(X - 1)} = \frac{\cancel{(X - 1)}(X^{d-1} + X^{d-2} + \cdots + X + 1)}{K\cancel{(X - 1)}},$$

  for a certain constant $K \in \mathbb{F}_r$.

The Barycentric Formula: Roots of Unity ii

- Since $L_0(1) = 1$, we can get the value of the constant

$$L_0(1) = \frac{\overbrace{1+1+\cdots+1}^{\text{d times}}}{K} = \frac{d}{K} = 1.$$

- Then, $K = d$ and we can express $L_0(X)$ as:

$$L_0(X) = \frac{X^d - 1}{d(X-1)}.$$

- Now, we can express the other Lagrange polynomials by means of the first one, displacing the values with the corresponding root of unity:

$$L_i(X) = L_0(X\omega^{-i}) = \frac{X^d - 1}{d(X\omega^{-i} - 1)}$$

The Barycentric Formula: Roots of Unity iii

- Therefore, the **Barycentric Formula** becomes

$$f(z) = \sum_{i=0}^{d-1} f_i L_i(z) = \frac{z^d - 1}{d} \sum_{i=0}^{d-1} \frac{f_i}{z\omega^{-i} - 1} = \frac{z^d - 1}{d} \sum_{i=0}^{d-1} \frac{f_i \cdot \omega^i}{z - \omega^i}.$$

- In this case, we dropped from the formula both the computation of $H(z)$ and the pre-computation of the $1/H(\omega^i)$ fractions.

EIP 4844 Figures

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

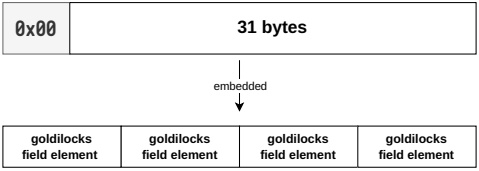
Outline

Feijoa Blob Data i

- Recall that, following the EIP4844 specification, the blob data will consist on 4096 elements f_i in $\mathbb{F}_r$, where $r = \texttt{0x73...01} \approx 2^{255}$ is the order of the embedded field in the BLS12381 curve.
- However, recall that the zkEVM is working over the field $\mathbb{F}_p$, where is the Goldilocks prime $p = 2^{64} - 2^{32} + 1$.
- Henceforth, each evaluation f_i encoded in the zkEVM will need to be embedded within a set of Goldilocks field elements, ideally optimizing space usage.

Feijoa Blob Data ii

- In seek of that, we require the most significant byte of each evaluation f_i to be `0x00`, that is, to have at most 248 bits.
- By doing that, we can fit f_i in 4 field elements.



- Dropping that condition, we should fit f_i in 5 field elements, one of them being almost empty, which is an unwanted scenario.
- From now on, the elements f_i satisfying that the most significant byte is `0x00` will be called **zkEVM-valid blob evaluation.**

zkEVM-invalid Blob Evaluation

- However, it can happen that the elements f_i sent have the most significant byte different from 0.
- The presence of one of such values should mark the whole blob **invalid**.
- Nevertheless, it remains necessary to compute $f(X)$ employing the barycentric formula based on the provided f_i, even if multiple of them are invalid.
- In this case, there are two concrete scenarios:
  - **Invalid BLS-Canonical Element:** The most significant byte lies in the set $\{\texttt{0x01}, \texttt{0x02}, \ldots, \texttt{0x72}\}$. This means that, even being invalid, the element represent a canonical representative of the BLS embedded field F_r.
  - **Invalid Non BLS-Canonical Element:** The most significant byte lies in the set $\{\texttt{0x73}, \texttt{0x74}, \ldots, \texttt{0xff}\}$. If this is the case, the element needs to be reduced to fit within $\mathbb{F}_r$ to ensure correct evaluation.

zkASM: $\mathbb{F}_r$ Arithmetic

- It has been introduced a new zkASM instruction that uses the Arithmetic state machine to check that

$$A \cdot B + C = op \mod D.$$

- Using this state machine, some subroutines has been introduced to provide the $\mathbb{F}_r$ arithmetic within the zkEVM generic registers, which recall that provide 256 bits divided in 8 subregisters of 32 elements each.

- `addFrBLS12381`: $\mathbb{F}_r$ addition.
- `subFrBLS12381`: $\mathbb{F}_r$ subtraction.
- `mulFrBLS12381`: $\mathbb{F}_r$ multiplication.
- `squareFrBLS12381`: $\mathbb{F}_r$ squarings.
- `invFrBLS12381`: $\mathbb{F}_r$ inversions.
- `reduceFrBLS12381`: Reduction from 256-bits to $\mathbb{F}_r$.
- `expBy4096FrBLS12381`: Exponentiation of z to 4096 modulo $\mathbb{F}_r$.

zkASM: Polynomial Evaluation over $\mathbb{F}_r$ using the Barycentric Formula i

- Using the subroutines specified before and the hardcoded roots of unity present in a zkASM file called `rootsOfUnity4096FrBLS12381` it is developed a subroutined called `polEvalFrBLS12381` which implements the barycentric formula to evaluate $f(X)$ at a given point z
- In the next slide we will explain the basic procedure implemented in the subroutine.
- However, it's important to note that in real zkASM code found here, the subroutine incorporates several optimizations aimed at reducing the number of steps and counters in some edge cases.

zkASM: Polynomial Evaluation over $\mathbb{F}_r$ using the Barycentric Formula ii

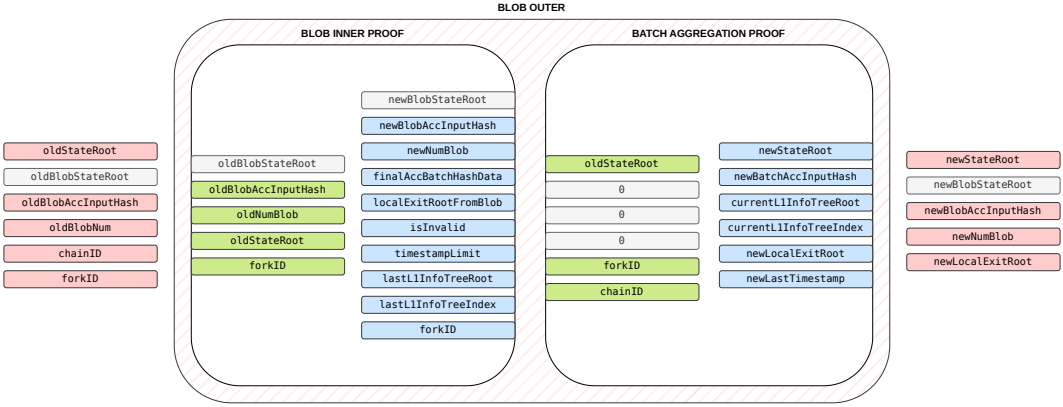
1. First of all, the z value at which we want to evaluate $f(z)$ is reduced to a element of $\mathbb{F}_r$ by taking its modulus.
2. Now, we take from the memory slot with label `polEvalFrBLS12381_index` the index `k` of the first element f_{k} within $f_0, f_1, \ldots, f_{d-1} = f_{4095}$ that is zkEVM-invalid (either canonical or non-canonical).
3. Now, from that index afterwards, we reduce all the appearing invalid non BLS-canonical elements, so that we can effectuate the evaluation properly, overwriting them in its memory position.
4. Now, it is time to compute $f(z)$ by using the barycentric formula

$$f(z) = \frac{z^{4096} - 1}{4096} \sum_{i=0}^{d-1} \frac{fi \cdot \omega^i}{z - \omega^i}.$$

blobOuter Circuit

- This circuit is in charge of aggregating a proof $\pi_{\texttt{re2}}$ (or $\pi_{\texttt{re1}}$ in the edge case of "aggregating" a single batch) and a proof $\pi_{\texttt{blobInner}}$ 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.

blobOuter Circuit: Inputs and Outputs



`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

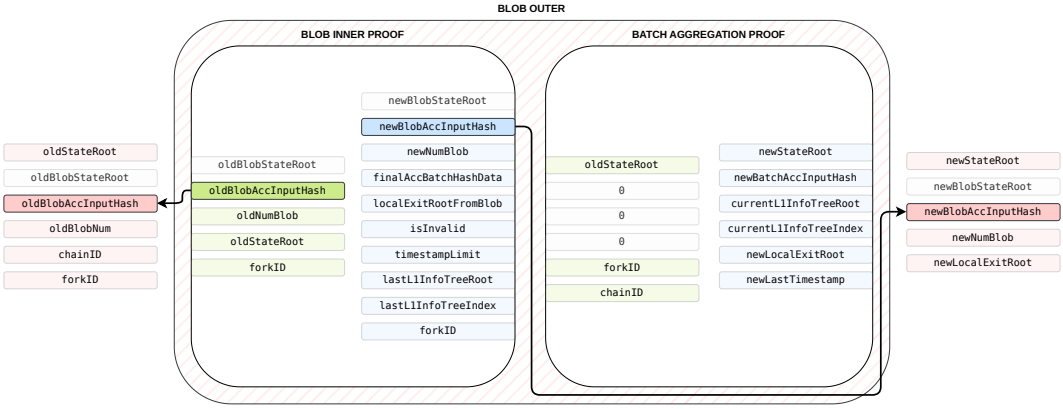
$$\texttt{isValid} = 1 \iff \begin{aligned} \texttt{isValidBlob} &= 1 \text{ and} \\ \texttt{isValidTimestamp} &= 1 \text{ and} \\ \texttt{isValidL1InfoTreeIndex} &= 1. \end{aligned}$$

Accumulated Blob Input Hash Integrity Check i

- Recall that, when proving batches we had a recursive cryptographic pointer called **accumulated input hash**, computed in the ROM from the batch input data, that was introduced in order to be able to proof that the batch input data within the proof aggregation is correct and that batches are proved aggregated in a correct order.
- Adopting a similar approach as with blobs, we introduce a blob accumulated input hash that is computed in the `blobInner` ROM and checked its integrity in the aggregation circuits.
- The blob accumulated input hash is computed recursively as follows:

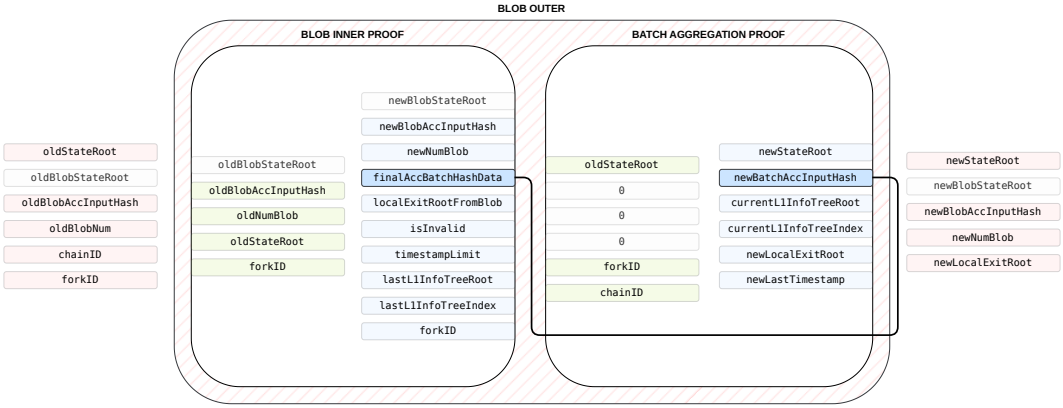
$$\begin{aligned}\texttt{newBlobAccInputHash} = \texttt{keccak256}(&\texttt{oldBlobAccInputHash}, \texttt{lastL1InfoTreeIndex},\\ &\texttt{lastL1InfoTreeRoot}, \texttt{timestampLimit}, \texttt{sequencerAddress}, \texttt{zkGasLimit}, \texttt{type}, \texttt{z}, \texttt{y},\\ &\texttt{blobL2Hashdata}, \texttt{forcedHashdata})\end{aligned}$$

Accumulated Blob Input Hash Integrity Check ii



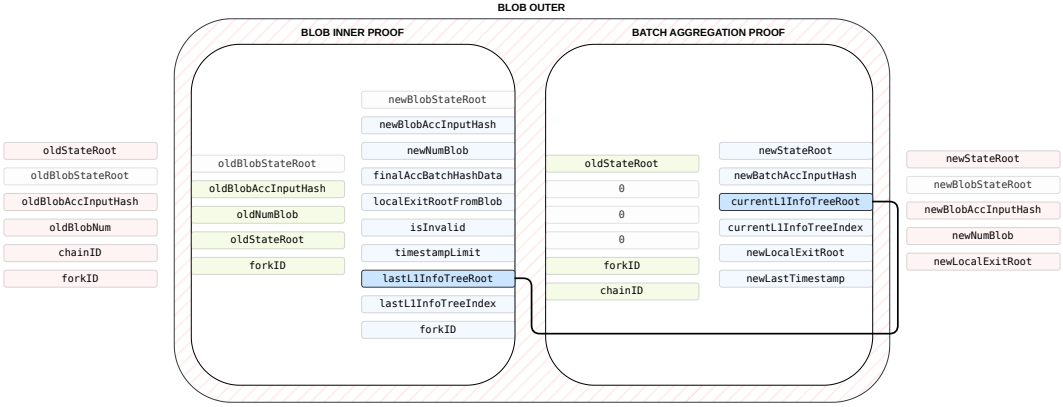
Batch Data Integrity Check

- Component-wise assertion that `finalAccBatchHashData`, computed in `blobInner`, coincides with `newBatchAccInputHash`, computed within the batch aggregation.



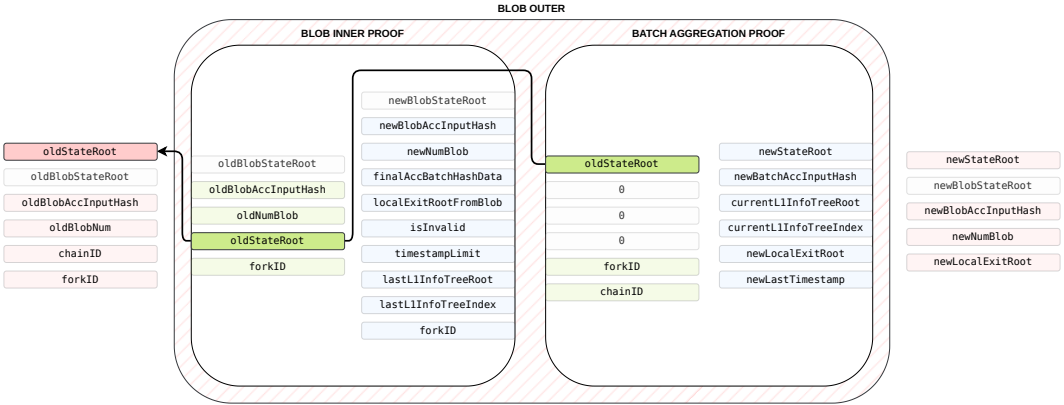
L1InfoTree Root Integrity Check

- 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`.



Old State Root Integrity Check and Output Assignment

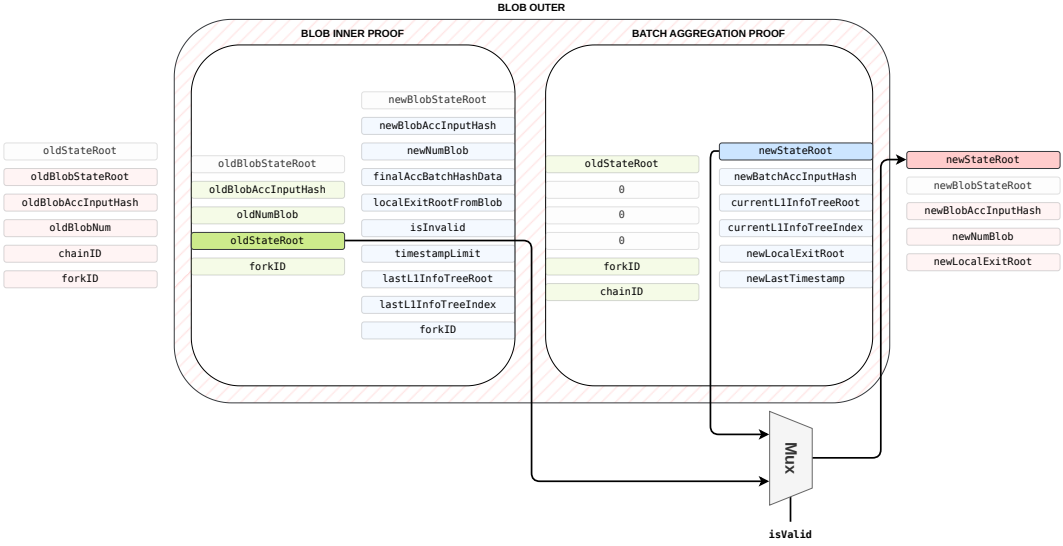
- Component-wise assertion that the old state root obtained from the blob data matches the old state root retrieved from the batch aggregation.
- Furthermore, the output of the old state root from the blob outer circuit is assigned to be one of these roots. This check is performed in all scenarios.



New State Root Integrity Check and Output Assignment

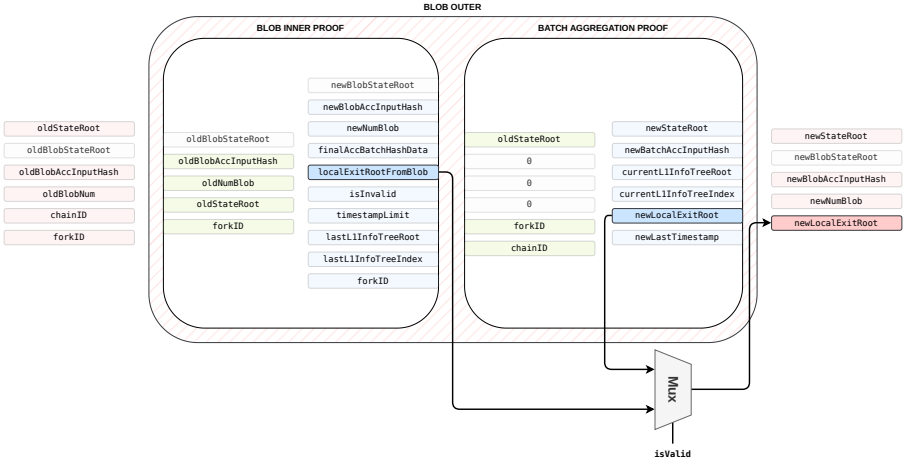
- The circuit's output signal `newStateRoot` is properly determined using a multiplexer.
- When the `isValid` signal is 0 the state root cannot be updated due to some errors.
- In such scenarios, it should be assigned the value of `oldStateRoot` obtained from the `blobInner` circuit.
- It's important 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.

New State Root Integrity Check and Output Assignment Circuit



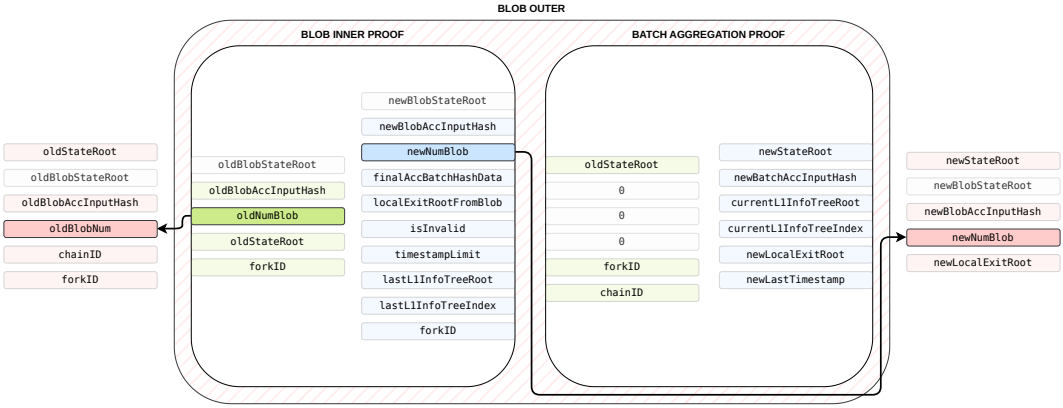
New Local Exit Root Update

- The circuit's output signal **newLocalExitRoot** is properly determined using a multiplexer.



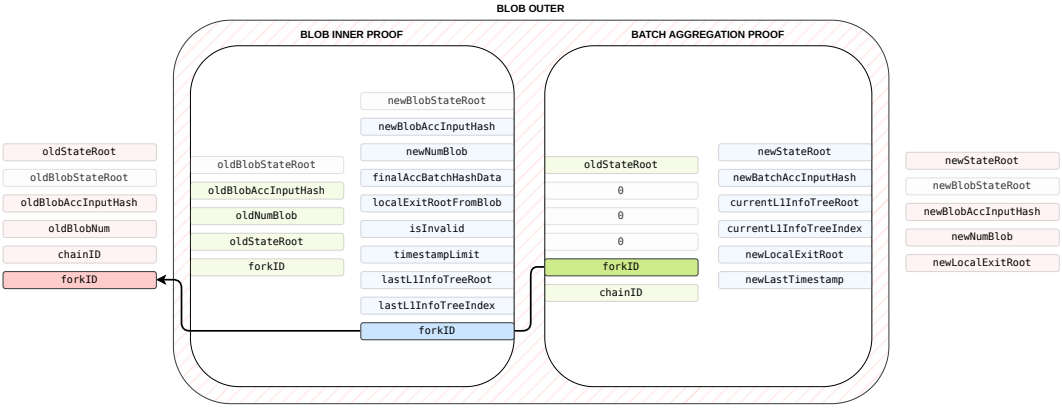
Output Blob Numbers Assignment

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



ForkID Integrity

- It is checked that the forkID from the blob data coincides with the one coming with the batch aggregation data.



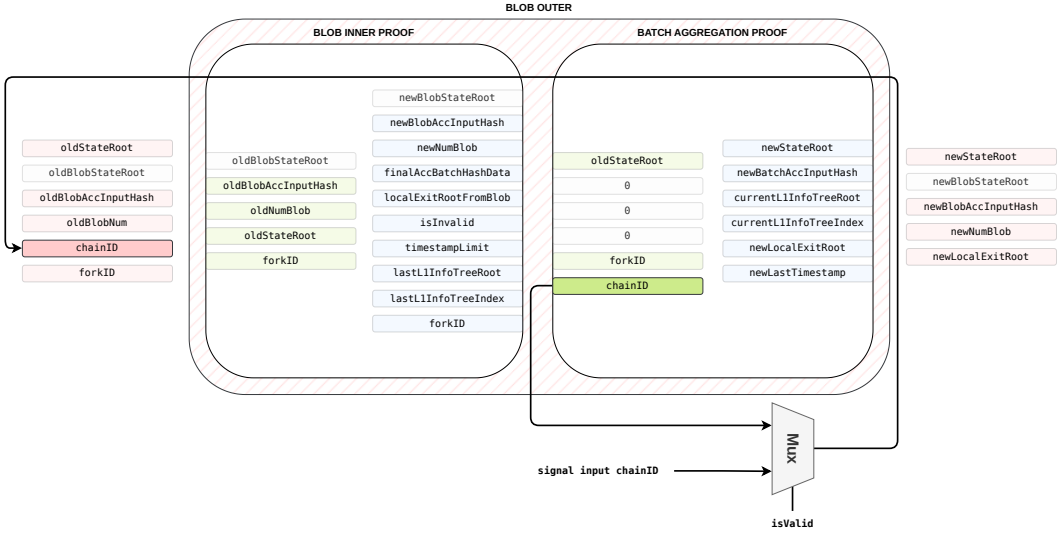
`chainID` Assignment i

- The `chainID` is a the most subtle public parameter to deal with.
- The `chainID` is assigned by the smart contract when the verification procedure is invoked.
- Thus, using an incorrect `chainID` during proving leads to verification failure, highlighting the importance of assigning the correct `chainID` to the output of the `blobOuter` circuit.

`chainID` Assignment ii

- If `isValid` signal is 1, we can retrieve this from the batch aggregation data, however, since `chainID` is not present in the public signals of the previous `blobInner` proof and can only be derived from the publics of the batch aggregation, a problem occurs if `isValid` signal is 0.
- In such scenarios, the prover's input may not be accurate for several reasons, rendering the batch aggregation data unusable and residual.
- To address this situation, the strategy involves passing the `chainID` independently as a private input to the `blobOuter` circuit.
- Since the `chainID` is hardcoded in the verification procedure by the smart contract, the prover cannot tamper with it.

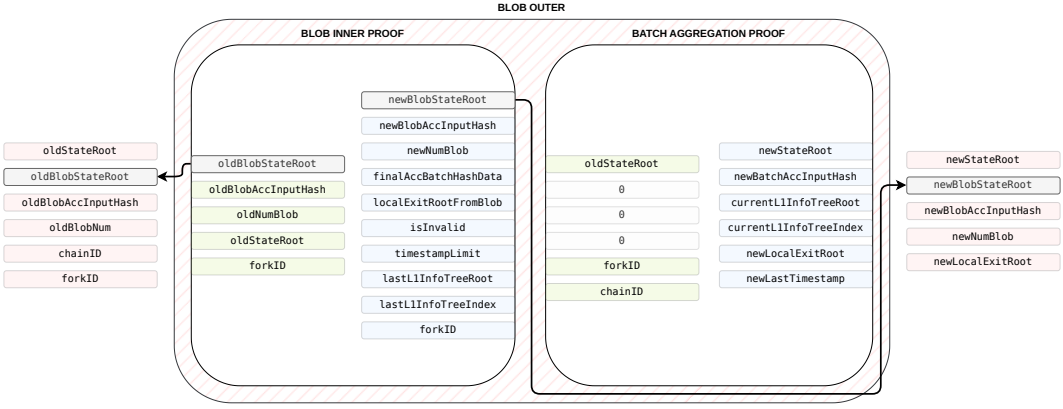
chainID Assignment Circuit



Compression-Related Unused Checks

- In order to avoid making a lot of changes to smart contracts in later forks, which would increase deployment efforts, the team has suggested including all the fields required for data compression in smart contracts within fork-feijoa.
- Thus, these fields must be incorporated into feijoa circuits as publics without currently restricting any specific (that is, correct) value (currently they are set to 0).

Compression-Related Checks



Considerations About the Constants

- In the whole aggregation process, we should consider whether we are aggregating just one batch or multiple batches because when aggregating only one batch, the `recursive2` circuit is not used.
- We must take this fact into account when incorporating the constants from the previous verifier into the batch's verifier circuit.
- Note that, since there's no aggregation of the `blobOuter` circuit, we can hardcode these constants.

- A multiplexer selects the verifier circuit's constant root based on `isOneBatch`, derived from the negation of `batch_isAggregatedCircuit`, indicating single or multiple batches.