The zkEVM Architecture

Part II: Sequencing and Proving

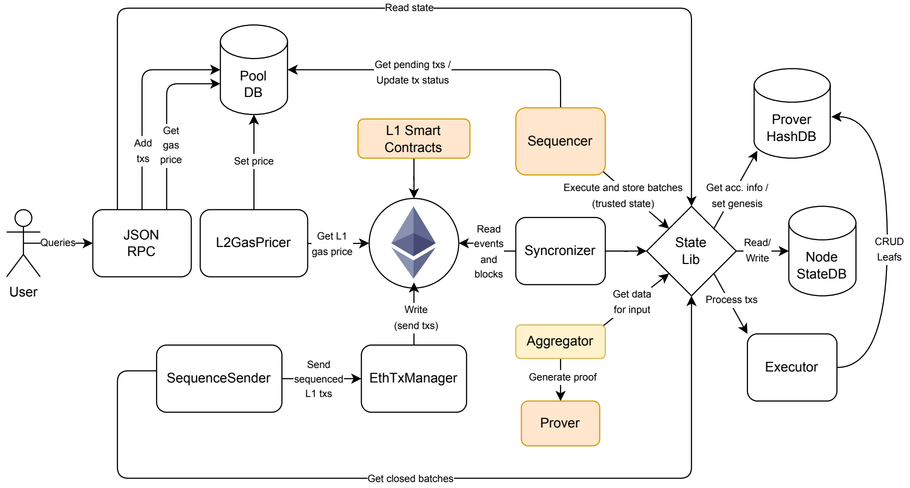
Polygon zkEVM & Universitat Politècnica de Catalunya (UPC)

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

December 20, 2023

Outline

Architecture of the zkEVM



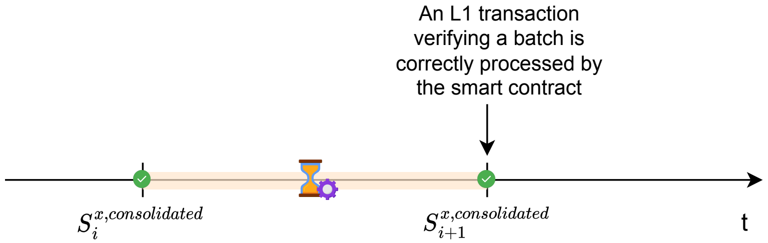
List of To Be Covered Concepts

- Consolidated state. ☐
- System's KPIs: throughput and delay. ☐
- Vertical and horizontal scaling. ☐
- Proofs aggregation. ☐
- Re-engineering the sequencer. ☐
- Sequence batches. ☐
- Virtual state. ☐

Consolidated (L2) State

Consolidated (L2) State

When a the L1 transaction submitting the proof π together with the corresponding publics is correctly executed, we say that a new L2 state is **consolidated** because at this moment the L2 state is proven to evolve correctly from $S_i^{L2_x}$ to $S_{i+1}^{L2_x}$ according to the batch processed.


An L1 transaction verifying a batch is correctly processed by the smart contract
$S_i^{x,consolidated}$
$S_{i+1}^{x,consolidated}$
t

zkEVM Key Performance Indicators (KPIs)

Delay

This is the delay from the moment that a user sends an L2 transaction until the effects of the execution of the transaction can be ”considered” part of the L2 state.

For user experience (UX), this is the main KPI.

Throughput

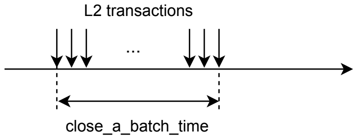
The throughput measures the system’s capacity of processing transactions.

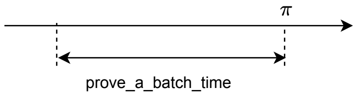
It can be computed in transactions per second, gas per second or batches per second.

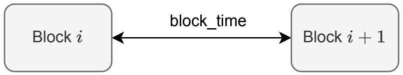
Let’s analyze and re-engineer our current system to improve these KPIs.

Parameters that Affect the KPIs

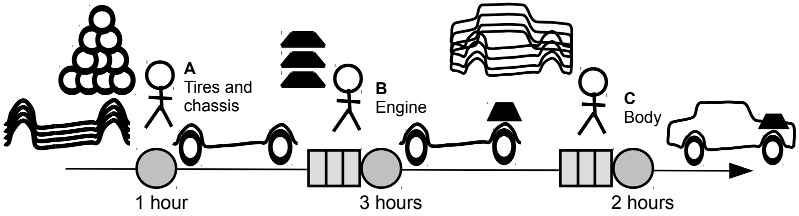
1. **`close_a_batch_time`**: this is the time that it takes get enough transactions to create (close) a batch or a given timeout for this purpose.
2. **`prove_a_batch_time`**: this is the time that it takes to generate the proof for a single batch (obviously, the size of the batch can affect this time).
3. **`block_time`**: this is the minimum time that it takes to execute L1 transactions.





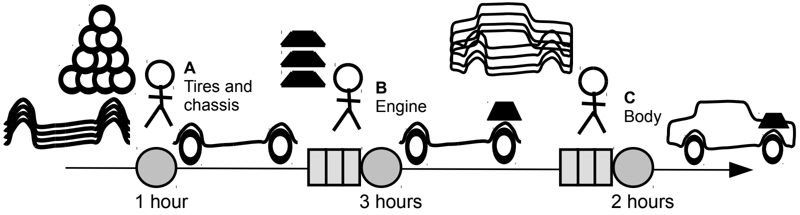


KPIs of a Pipeline i



- How much it takes to get the first car from the factory?
- How many cars per hour can we produce?

KPIs of a Pipeline ii

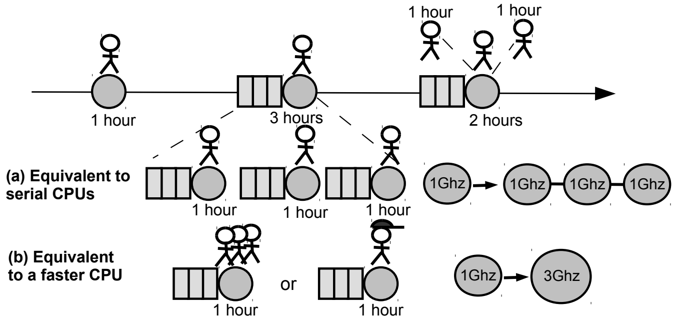


- How much it takes to get the first car from the factory? 1h+3h+2h=6h
- How many cars per hour can we produce (throughput)? 1/3h = 0.33cph

Enhance Throughput: Horizontal and Vertical Scaling in Pipelines

a) Add serial elements to the pipeline of small power (A.K.A **horizontal scaling**).

b) Add powerful elements (A.K.A **vertical scaling**).



Notice that horizontal scaling introduces more delay than vertical scaling.

However, in general vertical scaling is more costly to implement.

Approach: Close, Prove and Verify a Single Batch

- In this approach, we close a batch, generate the proof and then submit the L1 transaction to verify this batch.
- In this case we have the following KPIs:

$$delay = close\_a\_batch\_time + prove\_a\_batch\_time + block\_time \; [seconds]$$

$$throughput = \frac{1}{prove\_a\_batch\_time} \; [batches\, per\, second]$$

- **Remarks:**
  - Delay is a worst case regarding the creation of a batch and a best case regarding the execution of the L1 transaction for verifying the block.
  - For computing the throughput, we are considering that closing, proving and verifying a batch can be done in parallel with respect to other batches and that:

$$close\_a\_batch\_time << prove\_a\_batch\_time$$

$$block\_time << prove\_a\_batch\_time$$

$$block\_time = 15\,s\ (average) \quad prove\_a\_batch\_time = \ 120s\ (min) \quad close\_a\_batch\_time = 10\,s\ (max).$$

Improving the KPIs with Vertical Scaling

We want to increase the throughput and reduce the delay.

Our current limiting factor is the `prove_a_batch_time`.

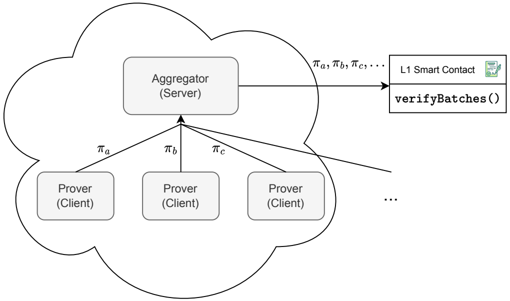
We can try to scale the system **vertically**:

- That is to say, generate the proof for a batch faster.
- However, this is in general costly, it requires either:
  - Run the provers in more powerful machines.
  - Optimize the proof system.
  - Or both.

Improving the KPIs with Horizontal Scaling (Naive)

Another option is to scale the system **horizontally**, the naive way of doing so is:

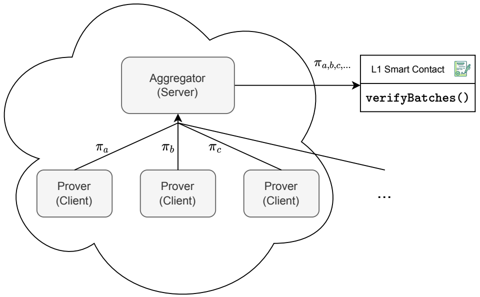
- We spin up multiple provers.
- Then, each prover sends its individual proof to an aggregator.
- The aggregator puts all these proofs into an L1 transaction and sends it to the smart contract so that batches can be verified.
- With this approach we have to close the batches in series but their proofs can be generated in parallel.
- Notice that the verification cost is proportional to the number of proofs sent.
- **Q.** Can we do it better?



Improving the KPIs with Horizontal Scaling (Proof Aggregation)

Another option is to scale the system **horizontally** with proof aggregation:

- We spin up multiple provers.
- Then, each prover sends its individual proof to an aggregator.
- Then, the aggregator is able to efficiently aggregate these proofs into a single proof that puts into an L1 transaction and sends it to the smart contract so that batches can be verified.
- Our cryptographic backend allows this approach since it allows to aggregate proofs.
- Notice that the verification cost in L1 is constant now.



KPIs with Aggregated Batches i

- Let's see which are the KPIs with aggregated batches.
- The throughput is (in batches per second):

$$\frac{N}{max(prove\_a\_batch\_time, \; N \cdot close\_a\_batch\_time, \; block\_time, \; aggregation\_time)}$$

- `aggregation_time`: is the time to aggregate the proofs of N batches (currently aggregation time is around 12 seconds).
- Observe that we can verify N batches taking the maximum of the periods of time of the denominator (since operations in the denominator can run in parallel for sets of N batches).
- In the case that the maximum time in the denominator is `prove_a_batch_time`, notice that we have increased the system's throughput by factor of N.

KPIs with Aggregated Batches ii

- Now, let's see which is the delay (in seconds):

$$delay = N \cdot close\_a\_batch\_time + prove\_a\_batch\_time + aggregation\_time + block\_time$$

- So, with the "straight aggregated batches approach", as you can observe, we are substantially increasing the delay regarding the "single batch approach".
- As mentioned, the delay is the main KPI with the UX, so:

  Q. How can we keep the gain in the throughput and enhance the delay?

  A. By splitting the batch processing in two steps: first order (sequence) and then, prove.

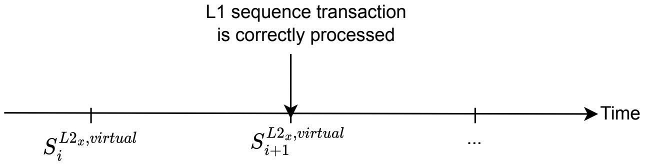
Enhancing Delay: Order, then Prove

- The idea behind **decoupling** the **batch sequencing (ordering)** process from the **batch proving** process is:
  a) Provide a low delay response to users about their L2 transactions.
  b) While being able to aggregate transactions to provide a high system throughput.
- Sequencing an L2 batch involves deciding the L2 transactions that will be part of the next batch (i.e., create or close the batch) and send them to L1.
- As the sequence of batches is written in L1, we have the data availability and immutability provided by L1.
- A sequenced batch will be eventually proved in the future.
- This means that the state after executing a sequenced batch will eventually (virtually) happen.
- This is why we call these states as **”virtual states”**.

Virtual (L2) State

Virtual (L2) State

The **Virtual state** is the state reached after executing a batch that has been sequenced in L1 but that is not proved yet. A virtual state is trustless since it relies on the L1 security.



- We can generate virtual states with low delay since, in general: `close_a_batch_time` $<$ `block_time`.

- Regarding the delay, now, we consider that it is the period of time since a user sends an L2 transaction until this transaction is in a virtual state:

$$delay^{(to\_virtual)} = close\_a\_batch\_time + block\_time$$

- Remark that our smart contract allows us to sequence multiple batches per L1 sequence transaction, in this case, $delay^{(to\_virtual)} = N \cdot close\_a\_batch\_time + block\_time$.

Summary of Approaches and KPIs

Approach	**Delay** [seconds]	**Throughput** [batches per second]
Single Batch	$delay^{(to\_consolidated)} = close\_a\_batch\_time + prove\_a\_batch\_time + block\_time$	$\frac{1}{prove\_a\_batch\_time}$
Aggregated Batches	$delay^{(to\_consolidated)} = N \cdot close\_a\_batch\_time + prove\_a\_batch\_time + aggregation\_time + block\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$
Virtual state	$delay^{(to\_virtual)} = N \cdot close\_a\_batch\_time + block\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$

Final Remarks

Advantages of decoupling sequencing batches from proving batches:

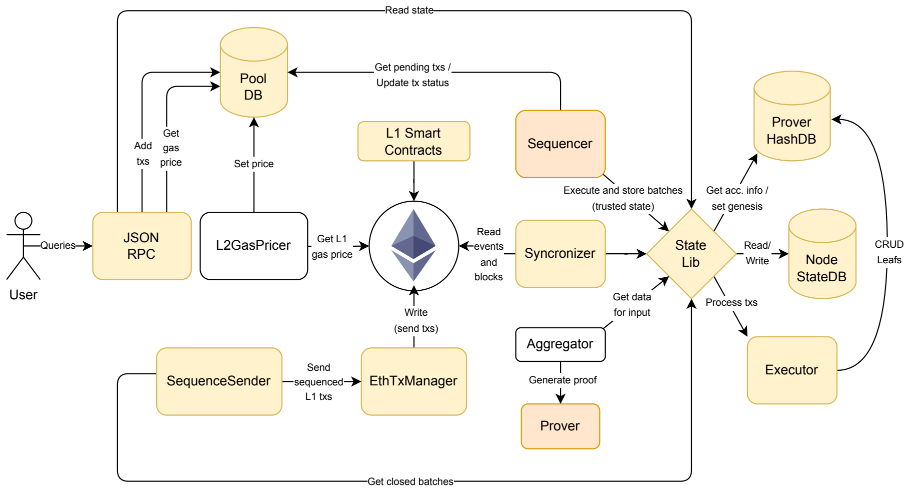
- Allows us to manage the queue of sequenced batches that are waiting to be consolidated.
- This involves introducing more or less delay and spinning up more or less provers.
- Thanks to the decoupling, this is done without impacting the perceived delay by users.

List of Covered Concepts

- Consolidated state. ☑
- System’s KPIs: throughput and delay. ☑
- Vertical and horizontal scaling. ☑
- Proofs aggregation. ☑
- Re-engineering the sequencer. ☑
- Sequence batches. ☑
- Virtual state. ☑

Outline

Architecture of the zkEVM: Sequencing Batches

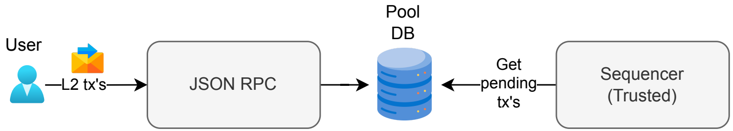


List of To Be Covered Concepts

- Trusted sequencer. ☐
- Batch pre-execution. ☐
- L2 coinbase address. ☐
- L2 Ether. ☐
- The cryptographic pointer `accInputHash`. ☐

Sequencing Batches

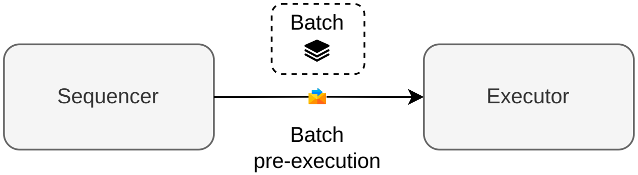
- Our current design has a **trusted sequencer**.
- But as we will see later on, we provide some anti-censorship mechanisms.



- The user sends L2 transactions, which will be stored in a database called the **pool**, through the JSON RPC.
- The sequencer selects a list of pending L2 transactions (not been yet sequenced) using some sort of criteria, and creates the batch to be sequenced and then proved.

Batch Pre-Execution

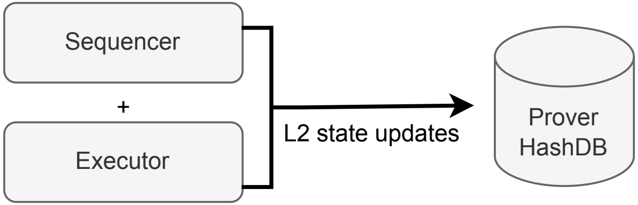
- In more detail, to create a batch, the sequencer first needs to do a **batch pre-execution** to make sure that the selected transactions within the batch:
  1. Fit in the execution traces available.
  2. Do not exceed the established gas limit.
- In this step, we need a **fast executor** (single-computation executor) that runs under `block_time`.



- When the transactions filling a certain batch are finally decided with a correct batch pre-execution, the sequencer writes batch in the node's StateDB as a **closed batch**.

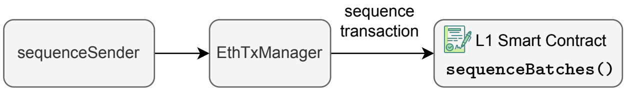
State Writing in the Prover’s HashDB

During the batch pre-execution, the sequencer also updates the Merkle tree of the zkEVM that is stored in the HashDB with L2 state changes.



Note. The throughput of the zkEVM highly depends on the speed at which we are able to close batches which is highly impacted by the batch pre-execution process.

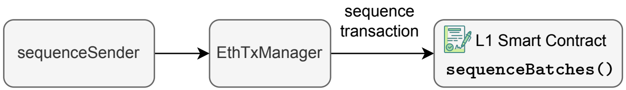
Send the Sequenced Batch to L1



- After the batch is closed, the sequencer stores the data of the batch into the node's StateDB.
- Then, the **sequenceSender** looks for closed batches and sends them to L1 the smart contract using the **EthTxManager**.
- This step provides L2 data availability in the L1 execution layer.
- In particular, the sequencer calls the **sequenceBatches** function in the L1 Smart Contract.

https://github.com/0xPolygonHermez/zkevm-contracts

Data within the Sequence Transaction

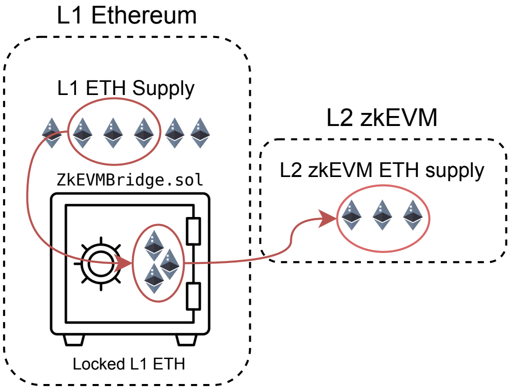


The calldata of the L1 ”sequence transaction” needs to contain the following information:

a) **L2 Transactions’ data:** An array containing the data for each batch, including all the contained transactions and a timestamp for its closing time.
b) The **L2 coinbase address**: which is the Ethereum address that will receive the fees paid by users.
c) A **timestamp** to show when the L2 transactions where sequenced.

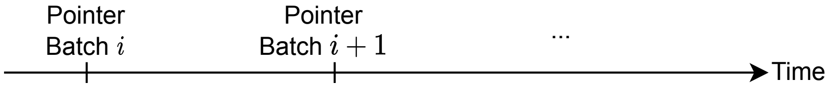
Comment About Fees in the zkEVM

- Regarding fees:
  1. The sequencer pays the fees of the sequence transaction in **(L1) ETH**.
  2. The coinbase address receives the L2 transactions fees of a certain batch in **L2 ETH**.
- The L2 ETH current supply is the amount of L1 ETH sent to L2.
- L1 ETH corresponds to L2 ETH in an 1:1 ratio.
- Note. More about how L1 ETH is moved to L2 or vice-versa will be explained later on when explaining the ”bridge”.



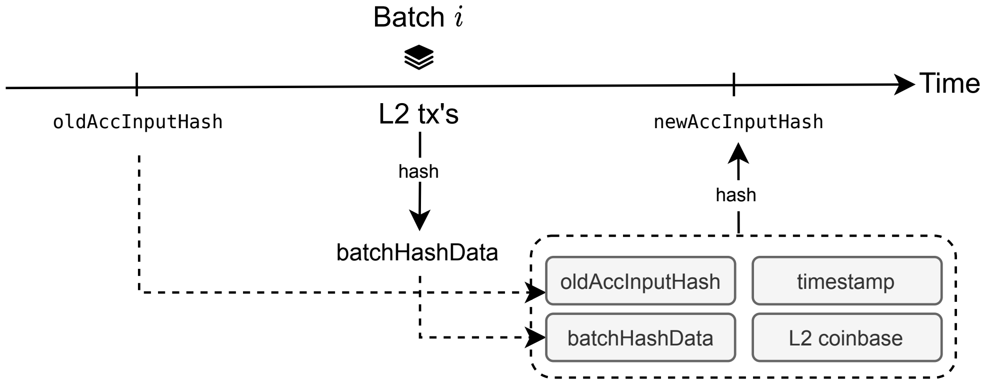
Accumulated Input Hash i

- The Smart Contract processes each L1 sequencing transaction and creates a “cryptographic pointer” for each batch.



- Each pointer unequivocally identifies a batch, its position and its data.
- These pointers are later used by provers to refer to the batch being proved and its data.

Accumulated Input Hash ii



Accumulated Input Hash iii

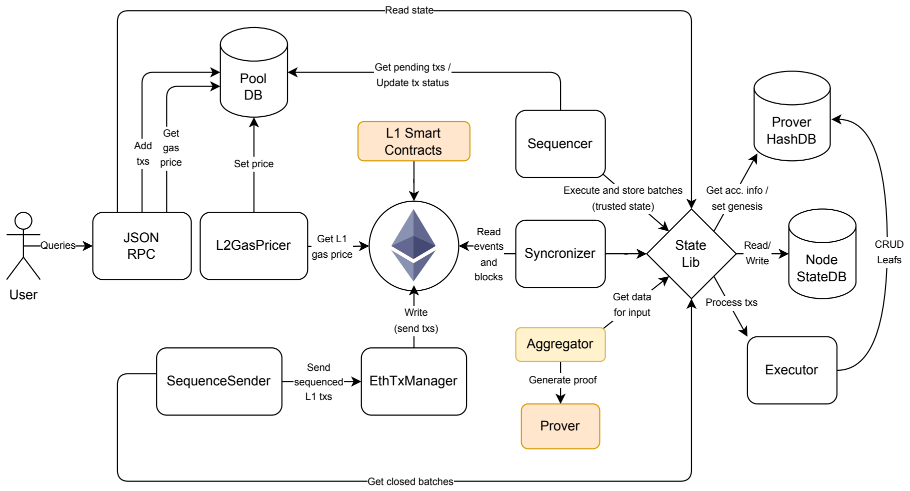
- Pointers are created by performing a KECCAK hash of:
  1. The previous pointer.
  2. The transactions included within the L2 batch.
  3. The batch timestamp.
  4. L2 coinbase.
- Since these pointers are created in a chained manner (observe that the current one includes the previous inside the inputs of the hash), they are known as *accumulated input hash* or `accInputHash`.
- The chained construction is necessary to ensure that data of batches is proved in the correct order.

List of Covered Concepts

• Trusted sequencer.	☑
• Batch pre-execution.	☑
• L2 coinbase address.	☑
• L2 Ether.	☑
• The cryptographic pointer `accInputHash`.	☑

Outline

Architecture of the zkEVM: Provers and Aggregators

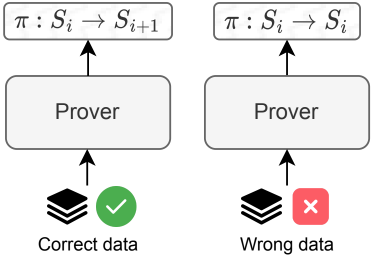


List of To Be Covered Concepts

- Prove anything paradigm. ☐
- Reverted transactions. ☐
- Invalid intrinsic transactions. ☐
- Invalid batch data. ☐
- zkCounters. ☐
- Out Of Counters (OOC) error. ☐
- OOC Batch. ☐
- VADCOPs: Variable Degree Composite Proofs (WIP). ☐
- Forced batches concept. ☐
- Recursion and aggregation. ☐
- The aggregator server. ☐
- Publics and privates re-design. ☐

“Prove Anything” Paradigm

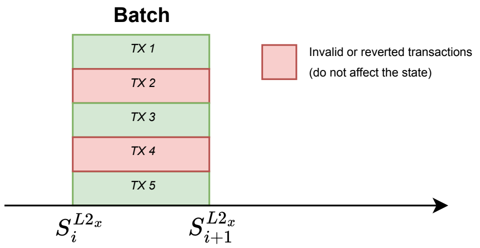
- We want to deal with potentially byzantine (malicious) sequencers.
- To do so, we follow the “prove anything” paradigm, which means that the prover can generate a proof of execution for any input data.
- The only requirement is that the batch needs to have a bounded amount of data (which is enforced by the smart contract when sequencing the batch).
- That is, the computation is always executed for any input data and, a proof is always generated either proving a **state change** or a **no state change**.



Generating the Proof: Errors in Transactions

1. **Reverted transaction.**
   A transaction may revert during its execution for several reasons (for example, out of gas, stack too big, revert called in code, etc) but this is a normal situation in the EVM processing.

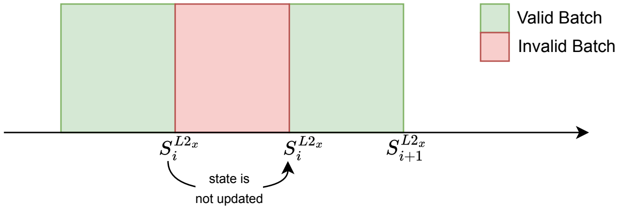
2. **Invalid intrinsic transaction.**
   Is a transaction that can't be processed and doesn't affect the state. Note that such a transaction could be included in a virtual batch. For example, incorrect nonce, not having enough balance, etc.



Generating the Proof: Errors in Batches

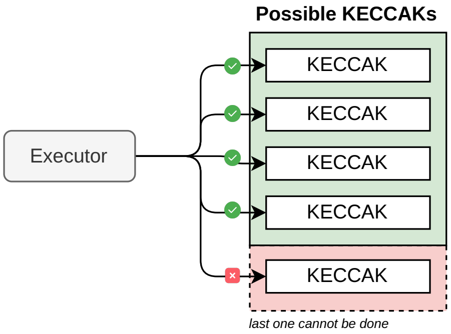
If the processing of a certain batch fails, then we do not update the state $S_{i+1} = S_i$ and we generate a proof for this **no state change**.

1. **Invalid data.**
   We can not decode transactions (RLP), so we have ”garbage input”.
2. **Prover resources exhaustion.**
   This is managed by the zkEVM with row counters (A.K.A zkEVM counters).



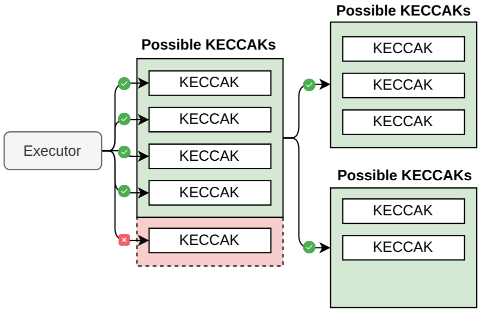
Brief on zkCounters and the OOC Error

- We use counters to count the rows of each State Machine, including the Main SM, we will use in an execution of a concrete batch.
- We manage these counters in the computation itself.
- In this way, we can prove that a batch does not change the state because while generating a proof we run out of resources.
- We refer to this error as an Out Of Counters (OOC) error.



VADCOPs: Variable Degree Composite Proofs (WIP)

- At the moment, there exists a backend limitation in the proving system that forces execution traces for all the States Machines to have the same amount of rows.
- However, this limitation will be dropped with the introduction of **VADCOPs: Variable Degree Composite Proofs,** which are currently under development.
- VADCOPs will allow us to split a large State Machine with a certain (big) amount of rows into smaller execution trances having fewer rows.
- This will introduce several advantages, the most significant one being the deletion of zkCounters for controlling prover resources.



Prove Anything and Forced Batches

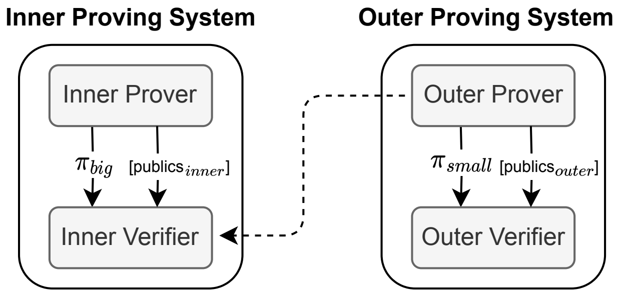
- The “prove anything” approach allows us to implement an anti-censorship measure called **forced batches**.
- Using these approach, an user can take the role of a sequencer for getting its L2 transactions into the virtual state in case the trusted sequencer is not doing so.
- The main use case is to allow a user to send bridge transactions in order to withdraw assets from L2 without the risk of censorship (which will make it impossible to withdraw the funds).
- Since it is an (untrusted) user who is sending the L2 batch data, we must be sure that we can prove anything that the user sends.
- The forced batches mechanism will be explained later on (when describing the different security measures implemented in the zkEVM).

Proof Recursion

Proof recursion

Proof recursion allows the prover to move from big proofs that are slow to verify into smaller and faster to verify proofs.

- Essentially, the prover of the next stage proofs that the verification of the previous stage is correctly performed.
- In general, this makes the final proofs smaller and faster to verify.
- Notice that we can also change the set of public values from one stage to the next one.

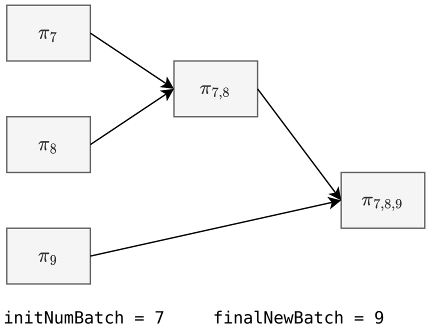


Proof Aggregation

Proof aggregation

Proof aggregation allows the prover to generate a single proof that covers multiple L2 batches.

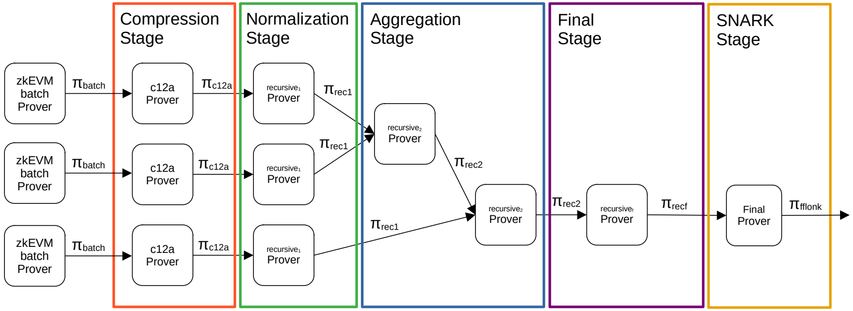
- With proof aggregation, we can send a single L1 transaction that aggregates multiple batches (improving the batch consolidation rate).
- The proving system also enforces that only consecutive batches can be aggregated.
- Notice that we can aggregate proofs that prove a single batch with other that prove multiple batches, this is thanks to a technique in the cryptographic backend that we call ”normalization”.
- Finally, remark that the smart contract also limits the maximum number of batches that can be aggregated.



zkEVM Recursion and Aggregation

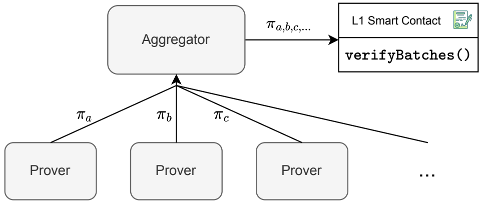
In the zkEVM, we implement recursion and aggregation.

The proof generation follows the following stages:



Introducing the Aggregator i

The **aggregator** is the service that will be in charge of performing the proof aggregation schema.

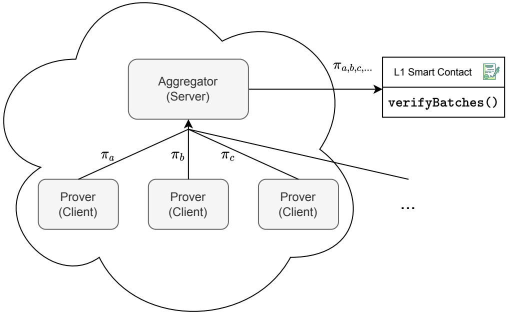


Calldata of the `verifyBatches` transaction:

- `initNumBatch`
- `finalNewBatch`
- `newStateRoot`
- Aggregated proof π

Introducing the Aggregator ii

- The aggregator is implemented as a network server to which provers, that act as network clients, connect to send their proofs.
- The aggregator decides how to (horizontally) scale provers to enable a proper batch consolidation rate (so that the queue of batches waiting to be consolidated does not grow too much).
- The aggregator manages a list of registered provers.
- The aggregator and the provers run in the cloud.
- Provers are high resource instances.



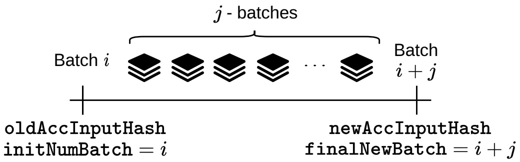
Aggregator Inputs and Outputs

Inputs:

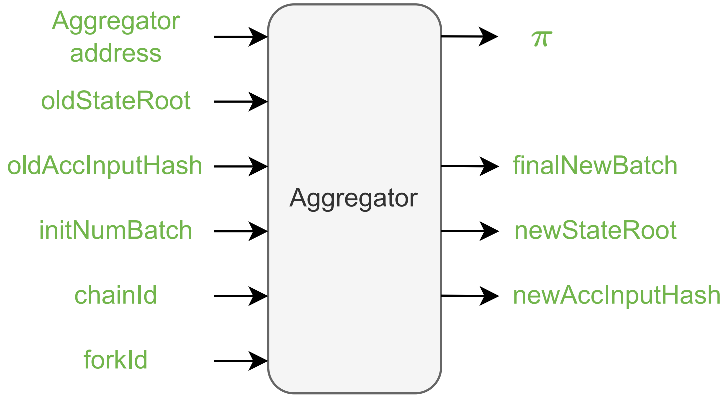
- **Aggregator address** (to avoid malleability in the verifyBatches transaction, i.e. to avoid that someone uses the proof of another aggregator.)
- `oldStateRoot`
- `oldAccInputHash`
- `initNumBatch` (In the verifyBatches tx)
- `chainId`
- `forkId`

Outputs:

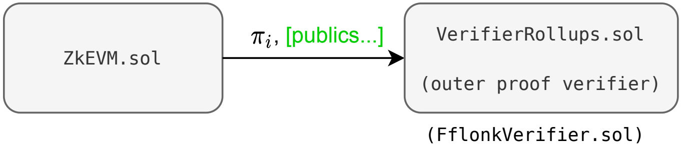
- `newStateRoot` (In the verifyBatches tx)
- `newAccInputHash`
- `finalNewBatch` (In the verifyBatches tx)



Aggregator Inputs and Outputs (Currently all are Public)



Verifier Contract

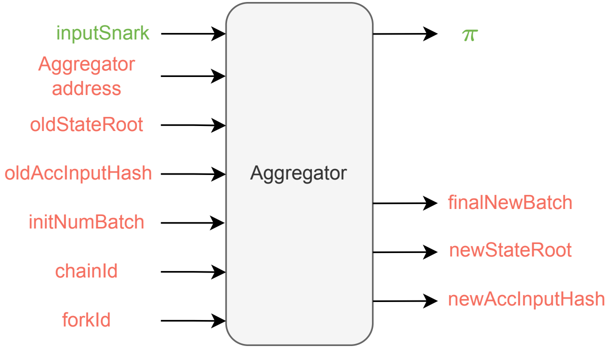


- This smart contract structure allows to upgrade or change the verifier if needed.
- Currently, we only provide a verifier called `FflonkVerifier.sol`.
- Note that with this design, the verifier has to use a list of publics, however, a verifier with a single input is cheaper.
- The trick is to have a single public that is the hash of all the previous publics and convert these publics into privates.

Aggregated Proof Inputs and Outputs Redesign i

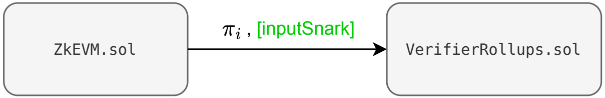
Now, for the **outer proof**, there will be a single public input in the aggregator called `inputSnark`.

- `inputSnark` is a KECCAK hash of all our previous public inputs and outputs:
  - Aggregator address
  - `oldStateRoot`
  - `oldAccInputHash`
  - `initNumBatch`
  - `chainId`
  - `forkId`
  - `newStateRoot`
  - `newAccInputHash`
  - `finalNewBatch`



Aggregated Proof Inputs and Outputs Redesign ii

- All the previous parameters are private inputs within the proof generation.



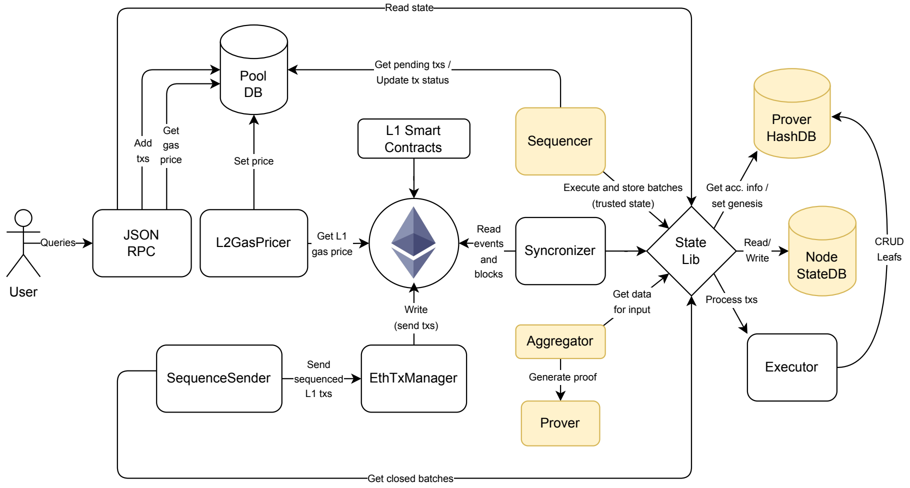
- Using this new approach, the proof system needs to check that:
    1. The hash of all private inputs (correctly ordered), actually matches **`inputSnark`**.
    2. The chained computation of the accumulated input hash of all the batches that the aggregator is processing matches the provided **`newAccInputHash`**.

List of Covered Concepts

- Prove anything paradigm. ☑
- Reverted transactions. ☑
- Invalid intrinsic transactions. ☑
- Invalid batch data. ☑
- zkCounters. ☑
- Out Of Counters (OOC) error. ☑
- OOC Batch. ☑
- VADCOPs: Variable Degree Composite Proofs (WIP). ☑
- Forced batches concept. ☑
- Recursion and aggregation. ☑
- The aggregator server. ☑
- Publics and privates re-design. ☑

Outline

Architecture of the zkEVM: zkEVM States



List of To Be Covered Concepts

- Trusted state. ☐
- Summary of approaches and KPIs. ☐
- State happy path for a batch. ☐
- Trusted state security. ☐

Trusted State

With our current design, if we sequence N batches, we can provide the following delay to users:

$$delay^{(to\_virtual)} = N \cdot close\_a\_batch\_time + block\_time$$

Q: Can we provide a smaller delay?

A: For this purpose, we will use the concept of **trusted state**.

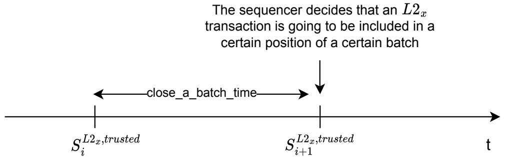
Trusted state

State reached when the sequencer closes a batch, that is, the sequencer decides the order of an L2 transaction in a certain position of a certain batch.

Delay to Trusted State

- The delay of an L2 transaction to reach the trusted state is the delay of closing a batch:

$$delay^{(to\_trusted)} = close\_a\_batch\_time$$


The sequencer decides that an $L2_x$ transaction is going to be included in a certain position of a certain batch
close_a_batch_time
$S_i^{L2_x,trusted}$
$S_{i+1}^{L2_x,trusted}$
t

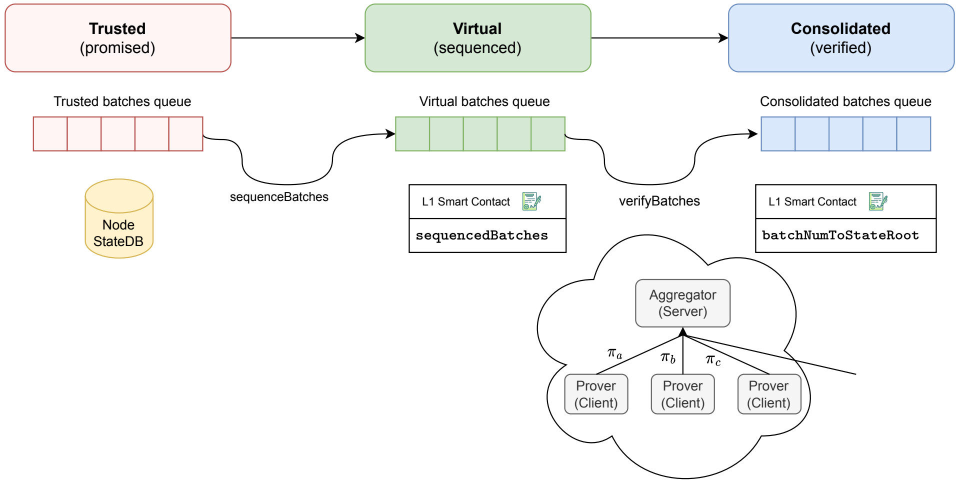
- The maximum delay to close a batch can be configured in the node in a variable called `timestampResolution`.
- Currently, the `timestampResolution` is set to 10 seconds, which is under the `block_time` but close to it to try to fill the batch.

Summary of Approaches and KPIs

Approach	**Delay** [seconds]	**Throughput** [batches per second]
Single Batch	$delay^{(to\_consolidated)} = close\_a\_batch\_time + prove\_a\_batch\_time + block\_time$	$\frac{1}{prove\_a\_batch\_time}$
Aggregated Batches	$delay^{(to\_consolidated)} = N \cdot close\_a\_batch\_time + prove\_a\_batch\_time + aggregation\_time + block\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$
Virtual state	$delay^{(to\_virtual)} = N \cdot close\_a\_batch\_time + block\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$
Trusted state	$delay^{(to\_trusted)} = close\_a\_batch\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$

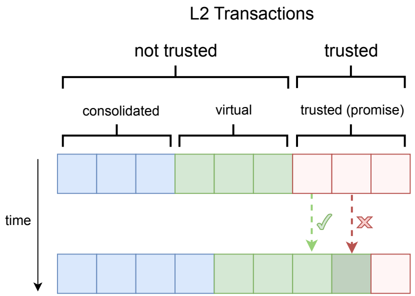
Note. The maximum value for N is fixed by the zkEVM smart contract and currently is set to `_MAX_VERIFY_BATCHES = 1000`

State Happy Path for a Batch



Security of the L2 Transactions in the Trusted State

- The trusted state is a ”promise” of the trusted sequencer.
- The trusted sequencer promises that it will send the trusted batch to L1 to convert this batch into a virtual batch.
- This state is considered ”trusted” because the trusted sequencer could commit to a certain sequence, but after send a different one to L1.
- From the UX point of view, transactions that imply a high value should wait until the virtual state is reached to be considered as immutable.

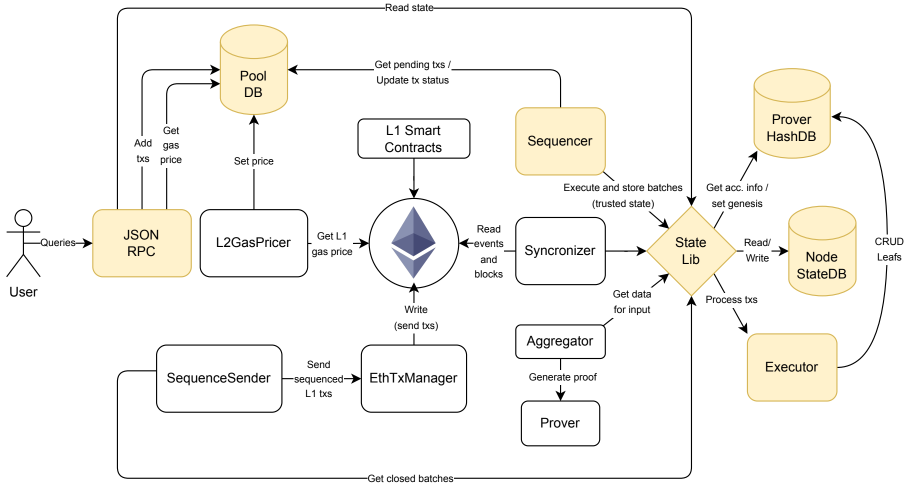


List of Covered Concepts

- Trusted state. ☑
- Summary of approaches and KPIs. ☑
- State happy path for a batch. ☑
- Trusted state security. ☑

Outline

Architecture of the zkEVM: RPC



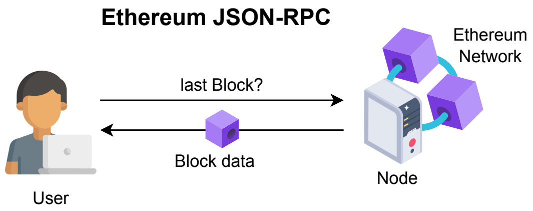
List of To Be Covered Concepts

- Review of the Ethereum RPC. ☐
- The zkEVM JSON-RPC. ☐
- L2 blocks. ☐
- Batches versus blocks. ☐
- The multiple blocks per batch approach (WIP). ☐
- Extra endpoints in the zkEVM JSON-RPC. ☐
- Public zkEVM RPCs and explorers. ☐
- Sending transactions. ☐
- Transaction checks before ingress the PoolDB. ☐
- Transaction estimation pre-execution. ☐
- Review of the sequencing batches process. ☐

Ethereum JSON-RPC

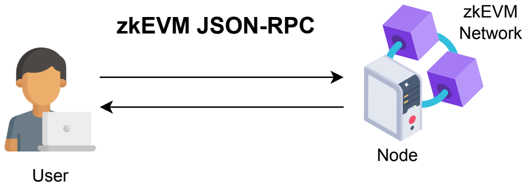
- The Ethereum JSON-RPC is a standard collection of methods that all Ethereum nodes implement and serves as the canonical interface between users and the network.
- As an example, we can ask for the last forged block as follows:

```
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["latest", false],"id":1}' localhost:8545
```



- The data of the block includes its transactions.

zkEVM JSON-RPC i

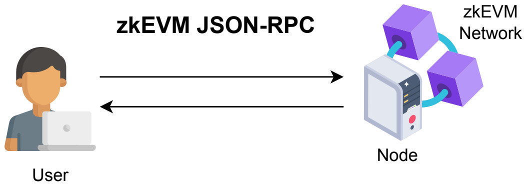


zkEVM JSON-RPC

Any Layer 2 equivalent with Ethereum must provide a compatible interface with the Ethereum JSON-RPC, implementing the same endpoints and getting similar responses. In the zkEVM architecture, this is what we call zkEVM JSON-RPC.

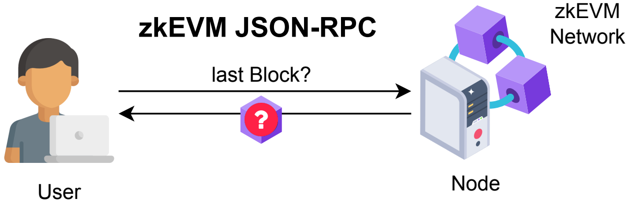
The zkEVM JSON-RPC implements all the methods coming from the Ethereum JSON-RPC but as we will see, **some responses of the zkEVM JSON-RPC have a meaning in the context of the zkEVM**.

zkEVM JSON-RPC ii



- In addition to implementing the methods from the Ethereum JSON-RPC, the zkEVM JSON-RPC also **introduces new methods** to deal with the zkEVM specifics.
- The complete set of supported endpoints, together with its current implementation state can be found in json-rpc-endpoints.md.
- The API specification for the specific zkEVM endpoints can be found in endpoints_zkevm.openrpc.json (which is defined in the defined in OPENRPC standard).

zkEVM JSON-RPC: Asking for the Last Block



- In Ethereum, when asking for the last block, the RPC answers with the last forged L1 block.
- In our case, we have batches that go through different states: trusted, virtual and consolidated.
- So, what should the zkEVM JSON-RPC answer if it is asked for the **last L2 block**?

Defining zkEVM L2 Blocks

- If for the definition of the L2 block we want to provide the minimum delay, and we have to choose between trusted, virtual or consolidated batches:
    - It is obvious that we should choose the last trusted batch as the last L2 block.
    - Recall that in this case, the delay is `close_a_batch_time`.
- However, a natural question is:

  Q. Can the zkEVM provide an even lower delay?

  A. The answer is yes if we create an L2 block before its corresponding batch is closed.
- In particular:
    - We can create L2 blocks that include transactions as soon as their order is decided, i.e. it is decided that they will go into a batch in a certain position.
    - The extreme situation, which is the one that we use in our current design, is that we create an L2 block for each ordered L2 transaction.
    - This provides the minimum delay possible that we call `order_a_transaction_time` since we do not have to wait for more transactions to close an L2 block.

L2 Batches and L2 Blocks

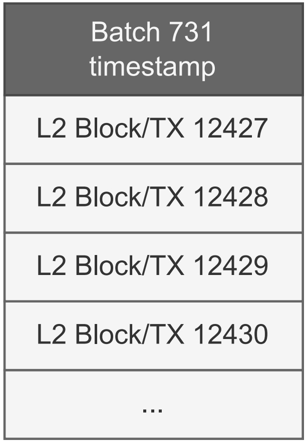
L2 Batch

A batch is the minimum amount of data for which we will generate a proof (that can be aggregated).

L2 Block

An L2 block is the data structure that is returned to the user through the RPC. L2 Blocks belong to L2 batches. As its associated batch, a block goes through all the zkEVM states (trusted, virtual and consolidated).

L2 Batches and L2 Blocks up to fork-dragonfruit (fork-5)



- Up to the dragonfruit-fork (fork-5) included, each L2 block contains a single transaction (same approach as Optimism).
- As a result, every batch contains as many blocks as transactions.
- The block is formed when it is decided that the transaction will be part of the trusted state.
- A **timestamp** is part of the batch data and this timestamp is shared for all the blocks.

Summary of Approaches and KPIs

Approach	**Delay** [seconds]	**Throughput** [batches per second]
Single Batch	$delay^{(to\_consolidated)} = close\_a\_batch\_time + prove\_a\_batch\_time + block\_time$	$\frac{1}{prove\_a\_batch\_time}$
Aggregated Batches	$delay^{(to\_consolidated)} = N \cdot close\_a\_batch\_time + prove\_a\_batch\_time + aggregation\_time + block\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$
Virtual state	$delay^{(to\_virtual)} = N \cdot close\_a\_batch\_time + block\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$
Trusted state	$delay^{(to\_trusted)} = close\_a\_batch\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$
L2 blocks	$delay^{(to\_L2\_block)} = order\_a\_transaction\_time$	$\frac{N}{max(prove\_a\_batch\_time, N \cdot close\_a\_batch\_time, block\_time, aggregation\_time)}$

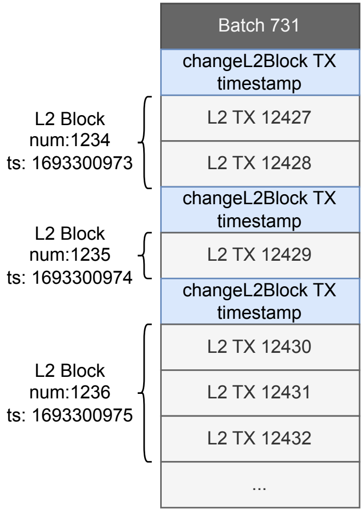
L2 Batches and L2 Blocks from fork-etrog (fork-6) i

- The approach of the fork-dragonfruit to manage L2 blocks creates though, problems:
  a) While the approach provides the minimum delay possible, it generates **a lot of data in the database due to the huge amount of L2 blocks created**.
  b) The approach lacks a way of **providing different timestamps for each block within a batch**, which breaks many Dapps that relay on this parameter to define the timing of the possible actions performed by the smart contracts.

- To address these issues, in fork-etrog:
  - Blocks with several transactions are enabled (this can be achieved by using a small timeout of seconds or miliseconds when creating the block for waiting for transactions).
  - The sequencer can change the timestamp of the different blocks within the batch by using a special transaction or marker called `changeL2Block`.

L2 Batches and L2 Blocks from fork-etrog (fork-6) ii

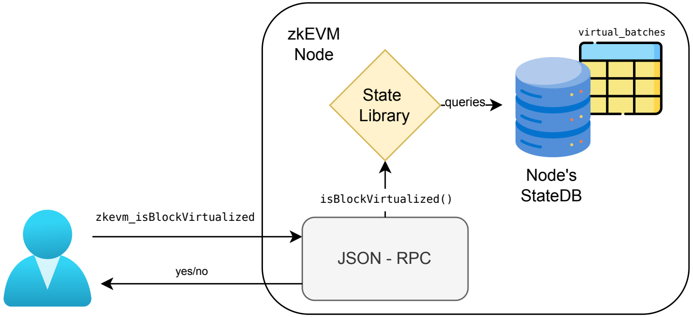
- As observed, a new special kind of transaction called `changeL2Block` is introduced in the batch to mark whenever there is a block change.
- The `changeL2Block` transaction is in charge of changing the `timestamp` and the L2 `blockNumber`.


Batch 731
changeL2Block TX timestamp
L2 Block num:1234 ts: 1693300973
L2 TX 12427
L2 TX 12428
changeL2Block TX timestamp
L2 Block num:1235 ts: 1693300974
L2 TX 12429
changeL2Block TX timestamp
L2 Block num:1236 ts: 1693300975
L2 TX 12430
L2 TX 12431
L2 TX 12432
...

Custom zkEVM Endpoints: Asking for L2 Block's Current State

Observe that for obtaining the last block, we use the same RPC call as in Ethereum.

However, the zkEVM protocol has extra endpoints to provide extra information about the state of the L2 block.



List of zkEVM-Specific RPC Endpoints

Recall that the list of specific zkEVM endpoints is in endpoints_zkevm.openrpc.json.

These endpoints are:

- `zkevm_consolidatedBlockNumber`: Returns the latest block number within the last verified batch.
- `zkevm_isBlockVirtualized`: Returns true if the provided block number is in a virtualized batch.
- `zkevm_isBlockConsolidated`: Returns true if the provided block number is in a consolidated batch.
- `zkevm_batchNumber`: Returns the latest batch number.
- `zkevm_virtualBatchNumber`: Returns the latest virtual batch number.
- `zkevm_verifiedBatchNumber`: Returns the latest verified batch number.
- `zkevm_batchNumberByBlockNumber`: Returns the batch number of the batch containing the given block.
- `zkevm_getBatchByNumber`: Gets the info of a batch given its number.

zkEVM RPCs and Block Explorers

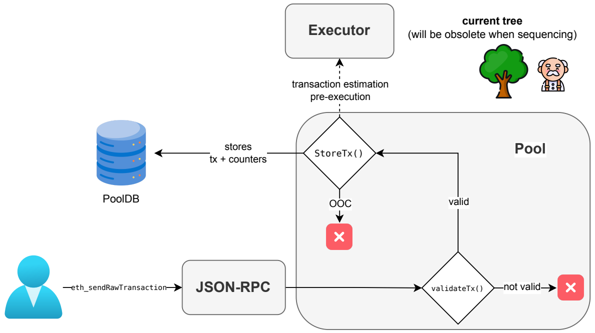
- Here we can find the URL of the Polygon zkEVM RPC and the chainID used for the testnet and the mainnet.
- Example:

```
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["latest", false],"id":1}' \
https://rpc.ankr.com/polygon_zkevm
```

- Here you have the block explorer of mainnet and here the block explorer of testnet.
- The block explorers are connected to the RPCs to obtain and then index the data of the chains.
- Also here we can find a list of alternative RPCs for the zkEVM mainnet.

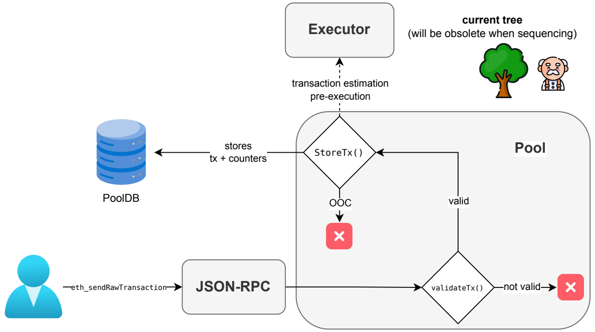
Sending a Raw Transaction i

- The figure shows the flow whenever a user calls the `eth_sendRawTransaction` endpoint to add some transaction into the **PoolDB**.
- In our architecture, the **rpc** component receives the L2 transaction from the user and, sends it to the **Pool** component.
- The **Pool** component is responsible for, among other things, adding transactions into the **PoolDB**.



Sending a Raw Transaction ii

- The **Pool** component performs some **preliminary validations** on the transactions, and if any of these validations fail, returns an error to the **rpc** component, which in turn returns the error to the user.
- The **Pool** component also performs a **transaction estimation pre-execution** with the current state root (which will be probably different from the state when the transaction is finally ordered).



Preliminary Validations to Insert Transactions in the Pool (`validateTx`) i

- A brief description of all the checks that `validateTx()` performs in `pool.go`:
    1. Checks that the transaction IP address has a valid format.
    2. Checks that the transaction fields are properly signed (in both current and pre-EIP-155). EIP-155 states that we must include the chainID in the hash of the to be signed data (which is an anti replay attack protection).
    3. Checks that the transaction's **`chainID`** is the same as the pool's **`chainID`** (which is the **`chainID`** of the L2 Network) whenever **`chainID`** is not zero.
    4. Checks that the transaction string has an encoding that is accepted by the zkEVM (more later).

Preliminary Validations to Insert Transactions in the Pool (`validateTx`) ii

5. Checks that the transaction sender's address can be correctly retrieved from the transaction, using the *ecRecover algorithm*.
6. Checks that the transaction size is not too large (more specifically, larger than 100132 bytes), to prevent DOS attacks.
7. Checks that the transaction's value is not negative (which can be the case since we are passing parameters over an API).
8. Checks that the transaction sender's address is not present in the **black list** (that is, is not blocked by the zkEVM network).
9. Checks that the transaction preserves the nonce ordering of the account.
10. Checks that the transaction sender account has not exceeded the maximum of transactions to have on the pool per user.

Preliminary Validations to Insert Transactions in the Pool (`validateTx`) iii

11. Checks that the pool is not full (currently, the maximum amount of elements in the queue of pending transactions is 1024).
12. Checks that the gas price is not lower than the set **minimum gas price** (this will be explained in more detail later in a section devoted to *economics*).
13. Checks that the transaction sender account has enough funds to cover the costs ($\texttt{value} + \texttt{GasPrice} \cdot \texttt{GasLimit}$).
14. The intrinsic gas of a transaction measures the required gas by the amount of transactional data plus the starting gas for the raw transaction (which is currently of 21000 or 53000 in case of being a contract creation transaction). If the provided gas is less than the computed intrinsic gas, the check is not passed.

Preliminary Validations to Insert Transactions in the Pool (`validateTx`) iv

15. Checks that the current transaction gasprice is higher than the other transactions in the `PoolDB` with the same `nonce` and `from`. This is because you cannot replace a transaction with another with less gasprice.
16. Checks that the sizes of the transaction's fields are compatible with the `Executor` needs, more specifically:
    - `Data` size: 30000 bytes.
    - `GasLimit, GasPrice` and `Value`: 256 bits.
    - `Nonce` and `chainID`: 64 bits.
    - `To`: 160 bits.

Remark. Some checks are guaranteed thanks to the typing system used in the code.

zkEVM Transaction Custom Encoding i

- Currently, **the zkEVM only supports non typed transactions** (check standard transaction encodings in Ethereum).
- The **to-be-signed-hash** is the same as in pre-EIP155 and EIP155 transactions so the signature is the same as in these transactions:

$$\begin{aligned} \textit{to-be-signed-hash}_{pre-EIP155} &= keccak(rlp(nonce, gasprice, startgas, to, value, data)) \\ \textit{to-be-signed-hash}_{EIP155} &= keccak(rlp(nonce, gasprice, startgas, to, value, data, chainid, 0, 0)) \end{aligned}$$

- However, the transaction string that we use for L2 transactions in the batches is a slight modification of the standard transaction string.
- The idea is to make it easier to the proving system to process L2 transactions.
- Let's consider the processing of a standard EIP155 transaction to our chain with $chainID = 1101$ and $parity = 1$.

zkEVM Transaction Custom Encoding ii

- We would receive the following transaction string:

$$rlp(nonce, gasprice, startgas, to, value, data, r, s, v = 2238)$$

- To check the signature, we have to rlp-decode the previous string to extract $nonce, gasprice, startgas, to, value, data, r, s$ and v.
- Then, we have to compute the *chainID* and *parity* from the v, in our example: $chainID = \lceil (v - 35)/2 \rceil = \lceil (2238 - 35)/2 \rceil = 1101$, $parity = v - (2 * chainID) - 35 = 2238 - (2 * 1101) - 35 = 1$.
- Then, we have to compute rlp-encoding of the following parameters:

$$rlp(nonce, gasprice, startgas, to, value, data, 1101, 0, 0)$$

- Compute the keccak hash:

$$to\text{-}be\text{-}signed\text{-}hash = keccak(rlp(nonce, gasprice, startgas, to, value, data, 1101, 0, 0))$$

- And finally, validate the ECDSA signature over the *to-be-signed-hash* with r, s with the *parity*.

zkEVM Transaction Custom Encoding iii

- As you may notice, there is a rlp-decoding and a rlp-encoding of the transaction parameters.
- To just do the decoding once, we can just create the transaction string as the *to-be-signed-hash* and concatenate the *r*, *s* and *parity* which are the only extra parameters required to verify the signature.
- According to this reasoning, we use the following **raw transaction string** for the zkEVM L2 transactions:

$$rlp(nonce, gasprice, gaslimit, to, value, data, chainId, 0, 0)|r|s|v|effectivePercentage$$

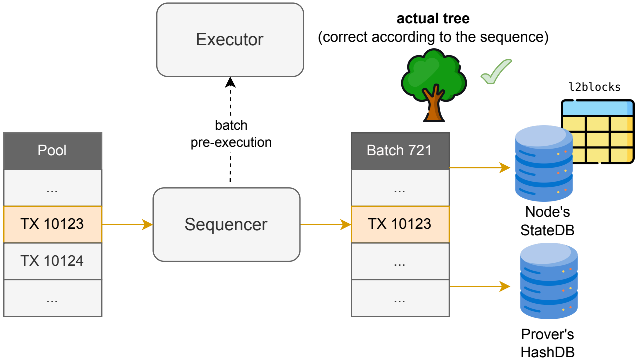
- The *parity* is expressed in the one-byte *v* parameter that can take the pre-EIP155 *parity* values (27 and 28).
- The parameter **`effectivePercentage`** which is a specific parameter of the zkEVM related to the fee system (by default we should use `0xFF`).
- Since the transaction signature is the same, it is easy to transform standard transactions into our custom format and vice-versa (here you JS utilities for doing these transformations).

Transaction Estimation Pre-execution (`StoreTx`)

- After previously checks are performed and no error has raised, another function called `StoreTx()` is called.
- This function performs a **transaction estimation pre-execution** (meaning that we are pre-executing with a probably wrong state root), that computes an estimations for zkCounters for pricing and prioritizing matters (we will deep into that later on).
- If we run Out Of Counters (OOC) at this precise moment, we raise an error and we do not add the transaction into the pool.

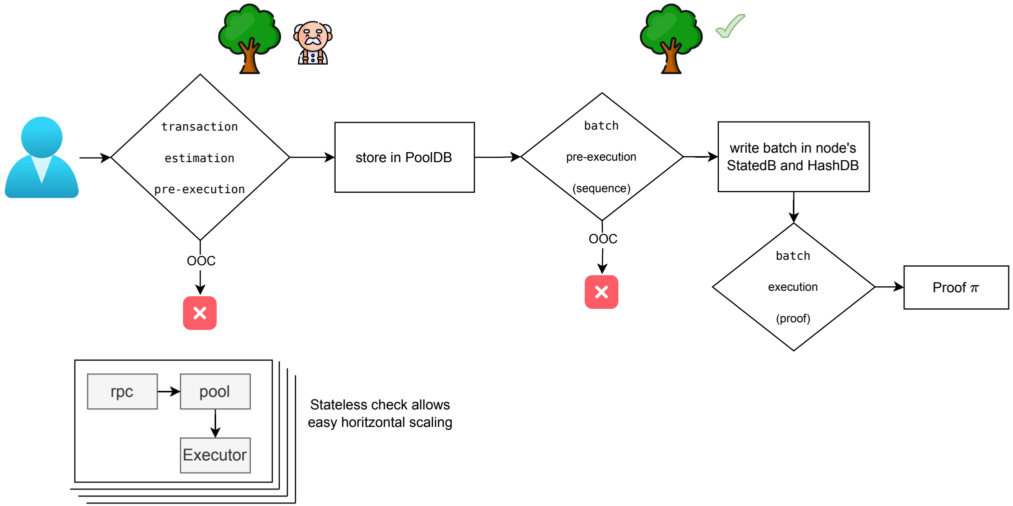
Sequencing Transactions from the Pool

The **sequencer** is in charge of selecting some transactions from the **PoolDB**, constructing a batch from them and adding them all into a specific table of the node's **StateDB** which keeps track of the L2 blocks. The **sequencer** also writes the **HashDB** with the L2 state.



Note. We will describe the transaction selection mechanism in more detail when describing the zkEVM *economics*.

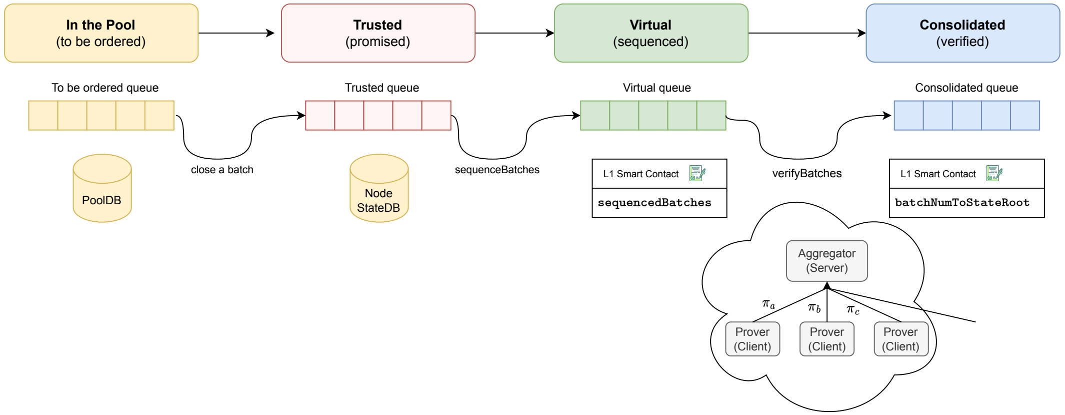
Pre-executions Summary



transaction estimation pre-execution
OOC
store in PoolDB
batch pre-execution (sequence)
OOC
write batch in node's StatedB and HashDB
batch execution (proof)
Proof π
rpc
pool
Executor

Stateless check allows easy horitzontal scaling

State Happy Path for a Transaction (Revisited)



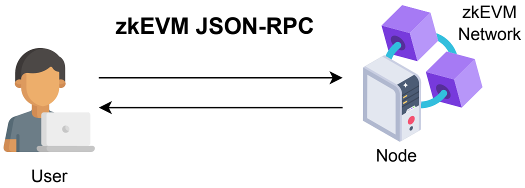
List of Covered Concepts

- Review of the Ethereum RPC. ☑
- The zkEVM JSON-RPC. ☑
- L2 blocks. ☑
- Batches versus blocks. ☑
- The multiple blocks per batch approach (WIP). ☑
- Extra endpoints in the zkEVM JSON-RPC. ☑
- Public zkEVM RPCs and explorers. ☑
- Sending transactions. ☑
- Transaction checks before ingress the PoolDB. ☑
- Transaction estimation pre-execution. ☑
- Review of the sequencing batches process. ☑

Outline

Massive Amounts of Queries via the zkEVM JSON-RPC i

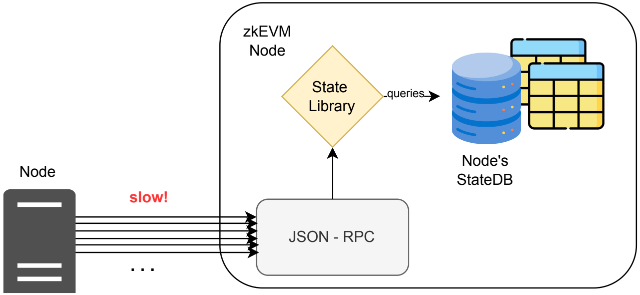
The zkEVM JSON-RPC is the natural way of obtaining information about the zkEVM and its data structures:



However, the JSON-RPC is not a fast way of obtaining massive amounts of L2 information.

Massive Amounts of Queries via the zkEVM JSON-RPC ii

In particular, accesses to the JSON-RPC require access to internal databases which makes the processing massive amounts of queries slow:



This happens when an external node[1] is synchronizing the whole state.

[1] We call such nodes **"permissionless nodes"**.

A Faster Way to Forward L2 Data: Streaming L2 Data

- In conventional L1 networks:

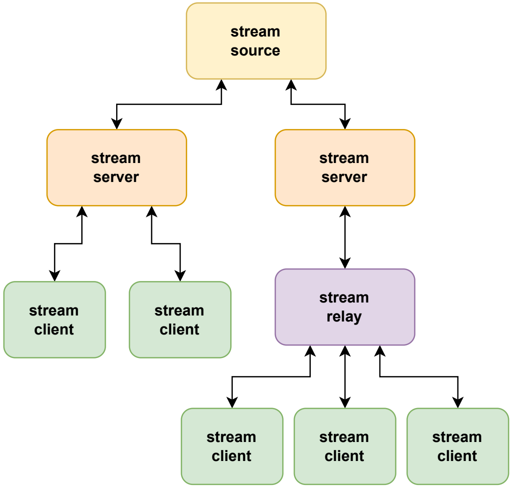
  The blocks are delivered to all the nodes in the network via some peer-to-peer protocol (typically called the ”Gossip” protocol and using algorithms like Kademlia).

- The idea is to do the same in the zkEVM:

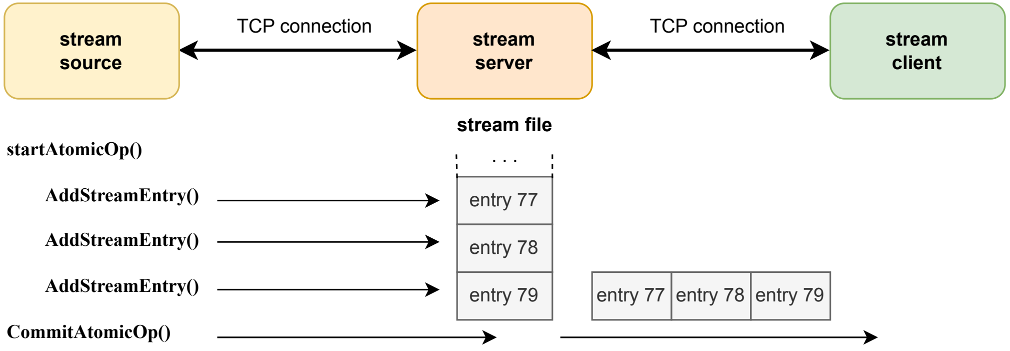
  Do not serve processed data via the JSON-RPC API to downstream nodes but just ”fast stream” the L2 data (batch data, block header data and transactions data) using some low overhead protocol.

Scalable Data Streaming

- To make the data streaming protocol scalable, it is not the zkEVM node itself who directly serves the data stream but an intermediate entity called ”data streamer”.
- The main zkEVM node serves the L2 data to the data streamer(s) who finally serve the data stream to end nodes or to intermediate relayers who act as clients of the upstream and servers for the downstream.
- This design creates a highly scalable data streaming infrastructure.



Entries and Atomic Operations i



We can also do a **RollbackAtomicOp()** to remove the operation's entries

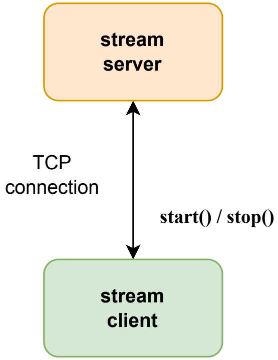
Entries and Atomic Operations ii

- The streaming protocol is built on top of TCP connections and it is used to transfer generic messages which in the protocol are known as ”entries”.
- All the entries of a flow are sequentially numbered starting from 0.
- One or more entries form an operation.
- Operations are atomic meaning that when the the unit that is sent
- We provide a library for the stream source to interact with the API.

  `https://github.com/0xPolygonHermez/zkevm-data-streamer`

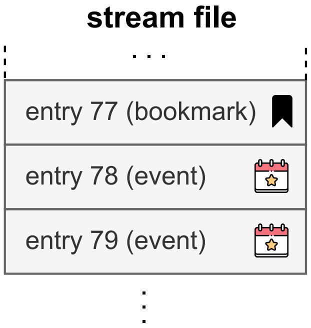
Streamer Client/Server Protocol: Basic

- **Start:** synchronizes from the entry number and starts receiving data streaming from that entry.
  The command format sent by the client:
  - `u64 command = 1`
  - `u64 streamType // e.g. 1:Sequencer`
  - `u64 fromEntryNumber`
  - If already started terminates the connection.
- **Stop:** stops the reception of the streaming transmission.
  The command format sent by the client:
  - `u64 command = 2`
  - `u64 streamType // e.g. 1:Sequencer`
  - If not started terminates the connection.



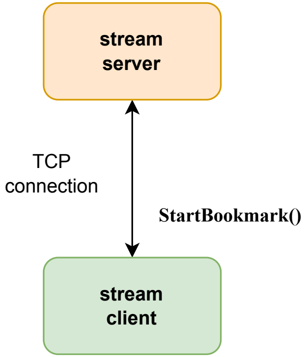
Events and Bookmarks

- Entries are identified by an `entryNumber`.
- Entries can be either events or bookmarks.
- **Events:**
    - Are defined by the application.
    - They provide relevant information about an event that deserves being streamed.



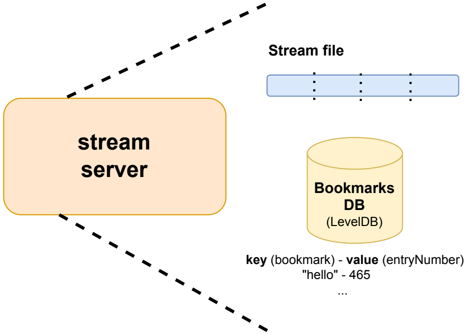
Bookmarks

- Are also entries and as such they have an `entryNumber`.
- Bookmarks link an `entryNumber` with a string of bytes that makes sense from a business logic point of view.
- Streamer clients can request to start the stream from a L2 block number, using the `StartBookmark` operation.
- The command format sent by the client for `StartBookmark` is:
    - `u64 command = 4`
    - `u64 streamType // e.g. 1:Sequencer`
    - `u32 bookmarkLength // Length of fromBookmark`
    - `u8[] fromBookmark`



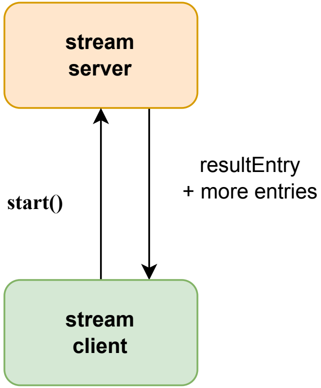
Bookmarks DB

- There is a key-value database (LevelDB) that maps the bookmark (used as a key) with its entry number (used as the value).
- When asked to start from a bookmark there is a binary search over the file to locate the bookmark and start the streaming.



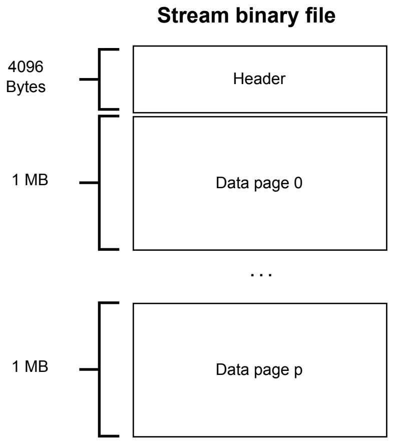
Streamer Client/Server Protocol: Result

- All the commands available for the stream clients return first a response, a Result entry.
- The format of the result entry is the following:
  - `u8 packetType // 0xff:Result`
  - `u32 length // Total length of the entry`
  - `u32 errorNum // Error code (0:OK)`
  - `u8[] errorStr`

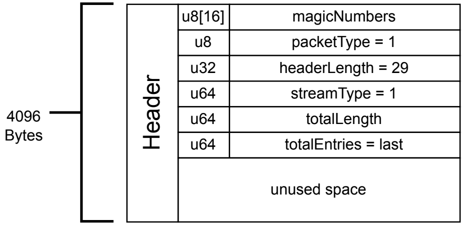


Stream File

- The file has a header and data pages.
- The header size is 4096 bytes.
- From the second page starts the data pages.
- Page size = 1 MB.
- If an entry does not fit in the remaining page space, the entry will be stored in the next page.



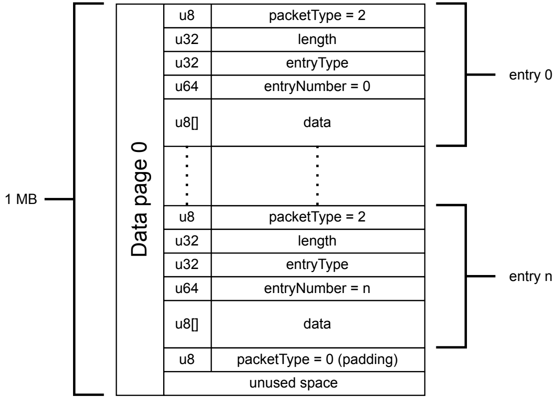
Stream File: Header



- `HeaderEntry` format:
- `u8 packetType = 1 // 1:Header entry`
- `u32 headerLength = 29 // Total header length`
- `u64 streamType // 1:Sequencer`
- `u64 TotalLength // Total bytes used in the file`
- `u64 TotalEntries // Total number of data entries`

Doing a rollback is just not update and doing a commit is write in the header the bytes.

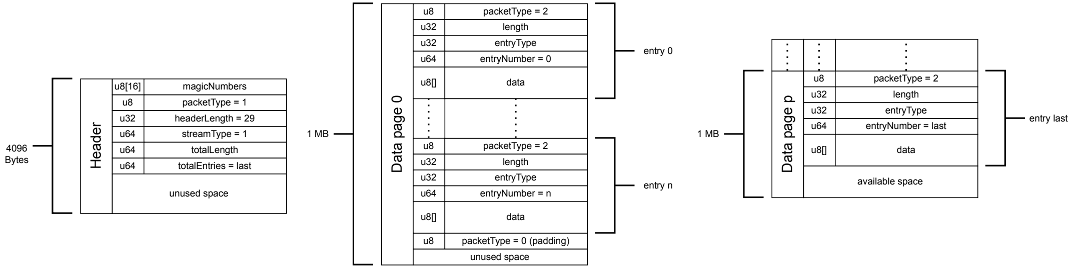
Stream File: Data Page



- `FileEntry` format:

```
· u8 packetType // 2:Data entry, 0:Padding
· u32 Length // Total length of data entry
             // (17 bytes + length(data))
· u32 Type // 0xb0:Bookmark, 1:Event1, 2:Event2,...
· u64 Number // Entry number (sequence from 0)
· u8[] data
```

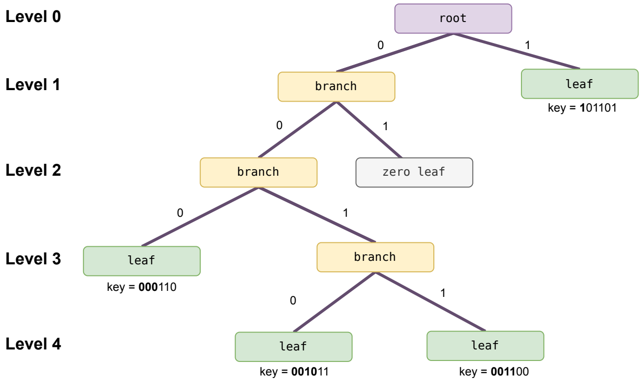
Stream File: Complete Format



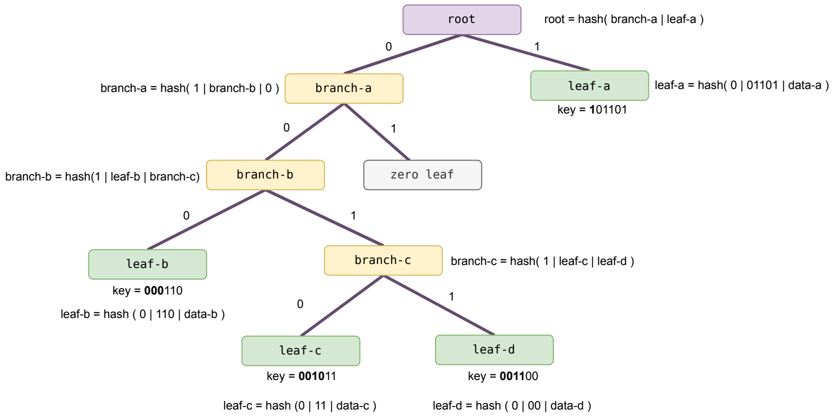
Outline

Trie Binary Sparse Merkle Tree

- The tree contains data for 4 keys: 000110, 001011, 001100 and 101101.
- Prefixes are used to place the `leaves` (data).
- Leaves are stored using 4 prefixes: **000, 0010, 0011** and **1**.
- `Branch` nodes are used to allow the navigation to leaves.
- Notice that we also need `zero` leaf nodes to show that there are no keys set under a prefix.

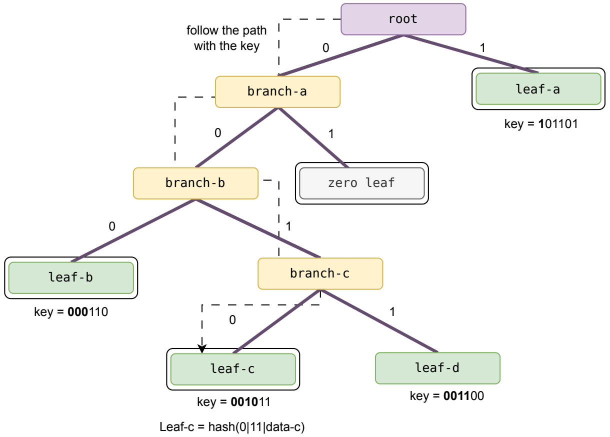


Tree Hashes


root
root = hash(branch-a | leaf-a)
0
1
branch-a
branch-a = hash(1 | branch-b | 0)
leaf-a
leaf-a = hash(0 | 01101 | data-a)
key = **1**01101
0
1
branch-b
branch-b = hash(1 | leaf-b | branch-c)
zero leaf
0
1
leaf-b
key = **000**110
leaf-b = hash (0 | 110 | data-b)
branch-c
branch-c = hash(1 | leaf-c | leaf-d)
0
1
leaf-c
key = **0010**11
leaf-c = hash (0 | 11 | data-c)
leaf-d
key = **0011**00
leaf-d = hash (0 | 00 | data-d)

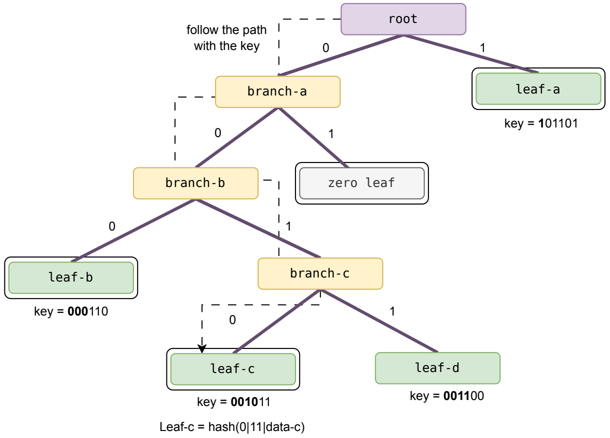
In addition, the empty tree is represented as $root = 0$.

Read a Key-Value: Inclusion Proofs



key: 001011 **proof:** ((11, data-c), leaf-b, 0, leaf-a)

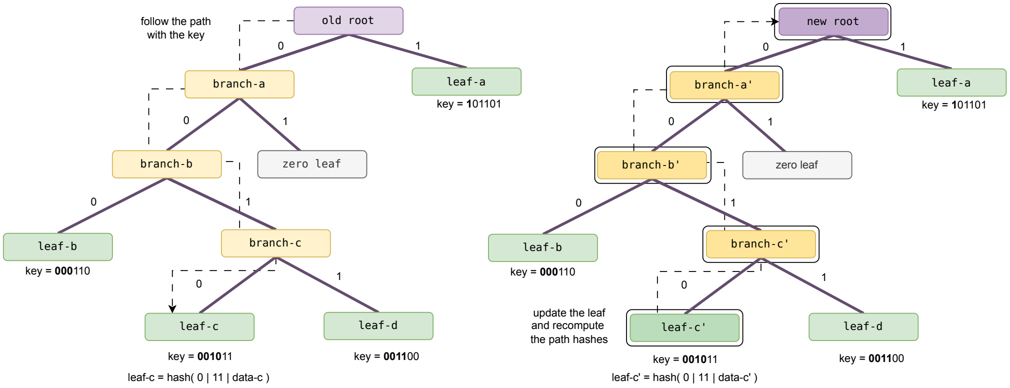
Read a Key-Value: Non-inclusion Proofs



Case1 (data leaf): **key:** 001000 **proof:** ((11, data-c), leaf-d, leaf-b, 0, leaf-a)

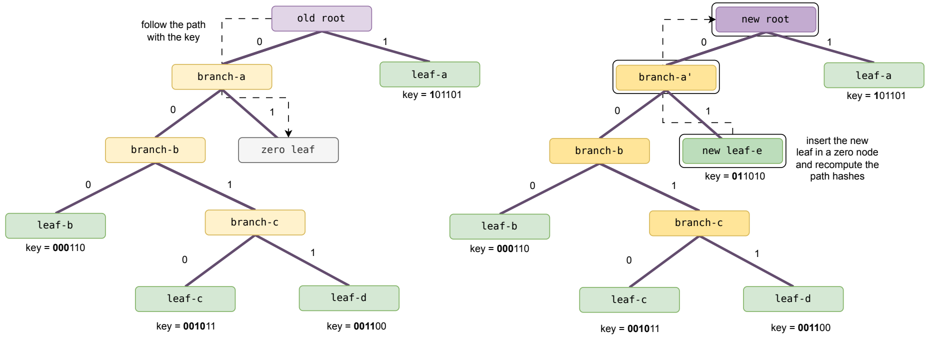
Case2 (zero leaf): **key:** 011001 **proof:** (branch-b, 0, leaf-a)

Update a Key-Value

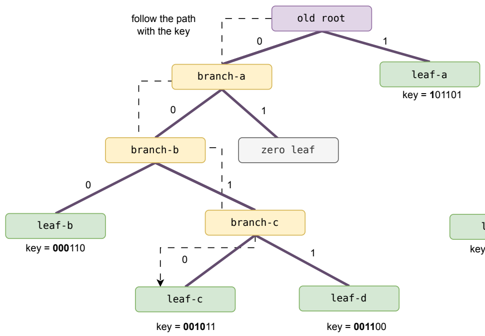


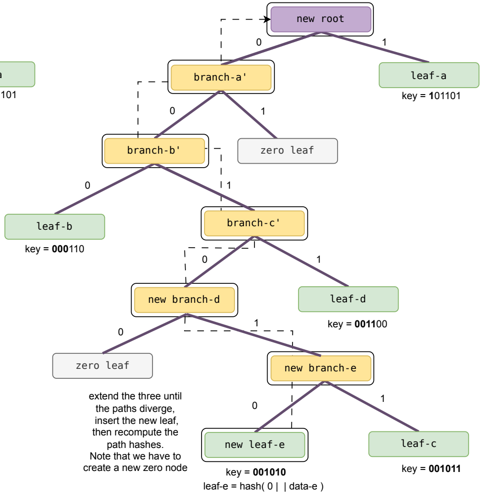
Note. When a value is set to 0, the default value, this is equivalent to remove the leaf (we show this case later).

Create a Key-Value i



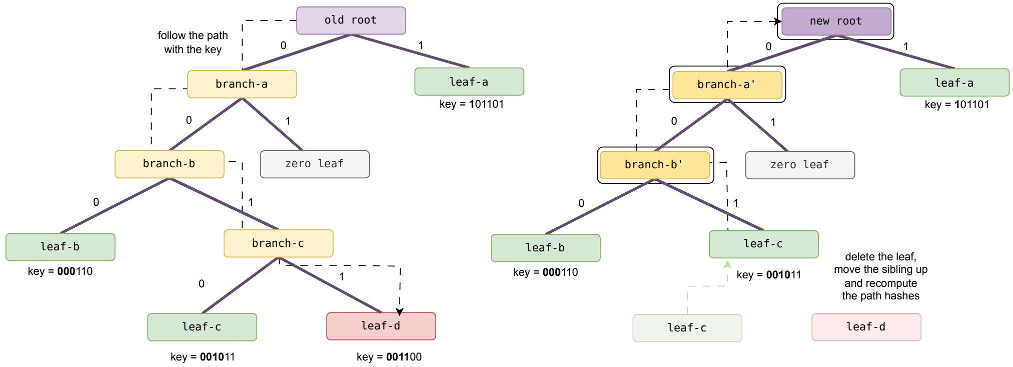
Create a Key-Value ii



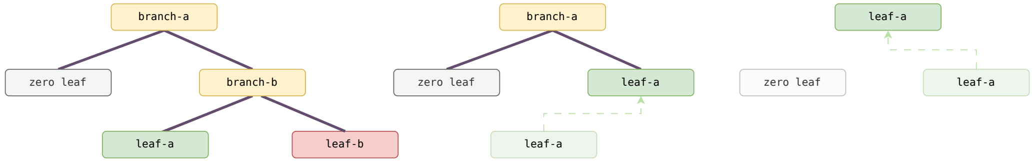


extend the three until the paths diverge, insert the new leaf, then recompute the path hashes. Note that we have to create a new zero node

Delete a Key-Value i

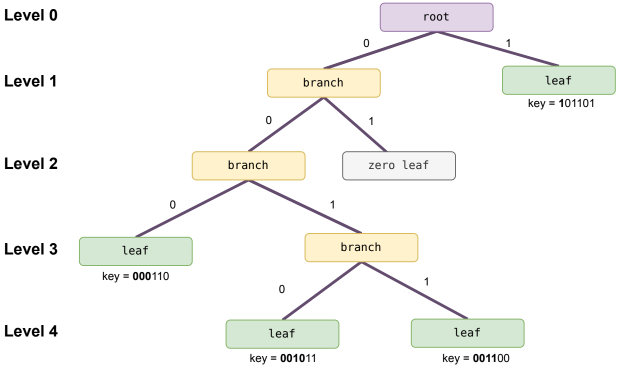


Delete a Key-Value ii



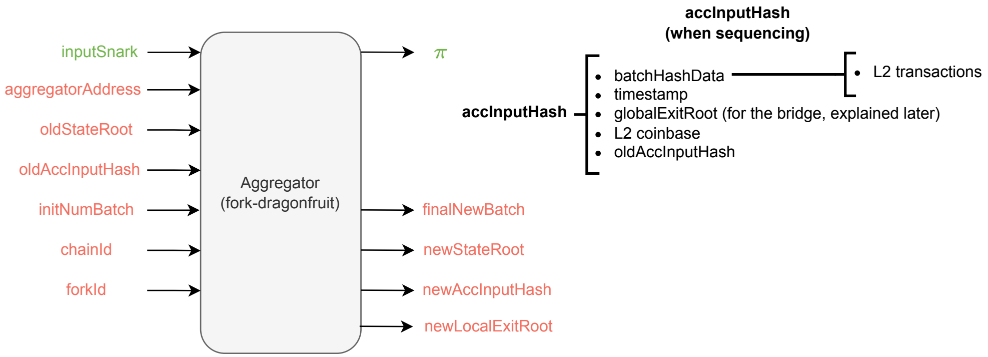
The Tree Features

- This Merkle tree is:
  a) Structure: key-value.
  b) Arity: binary.
  c) Unbalanced.
  d) Sparse.
  e) Trie/Prefix.
  f) Updatability: suitable for read-write operations.



Outline

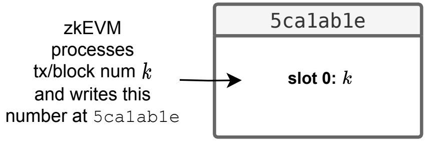
Sequencing and Proving up to fork-dragonfruit (fork-5)



Where is the transaction/block number stored?

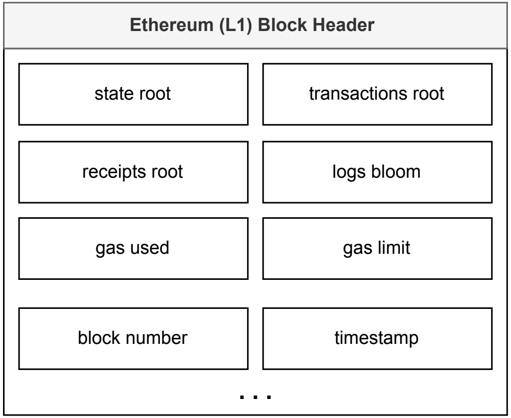
Transaction/Block Number at **`0x5ca1ab1e`**

- The last transaction/block number processed is stored in the L2 state.
- In more detail, the L2 system smart contract under the address `0x5ca1ab1e` stores at **slot 0** the number of the last block/transaction processed by the zkEVM.


zkEVM processes tx/block num k and writes this number at 5ca1ab1e
5ca1ab1e
slot 0: k

Blockhash in L1

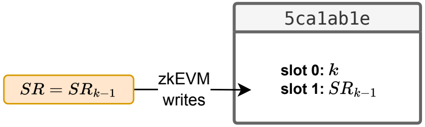
In the L1 EVM, the `BLOCKHASH Opcode` provides the hash of the Ethereum L1 block header which includes information like: coinbase address, blocknumber, timestamp, root of the state trie, root of transactions trie, root of receipt trie, difficulty, gas limit, gas used, logs, etc.



https://ethereum.org/en/developers/docs/blocks/#block-anatomy

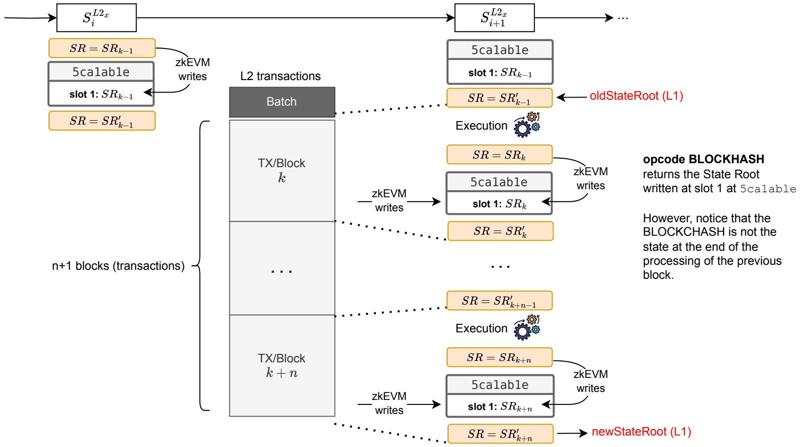
Blockhash in L2 up to fork-dragonfruit (fork-5)

- In fork-dragonfruit, we just provide the state root as `BLOCKHASH` in L2.
- At **slot 1** the `0x5ca1ab1e` contract the zkEVM stores the last (current) State Root after the processing of a block.
- When the `BLOCKHASH` Opcode is executed within an L2 transaction, the slot 1 of the `0x5ca1ab1e` system contract is provided.



opcode BLOCKHASH returns the State Root written at slot 1 at `5ca1ab1e`

Processing L2 Blocks up to fork-dragonfruit (fork-5)



opcode BLOCKHASH returns the State Root written at slot 1 at `5ca1ab1e`

However, notice that the BLOCKCHASH is not the state at the end of the processing of the previous block.

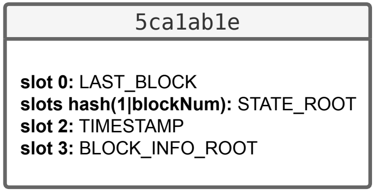
When all the blocks are executed, notice that the value at the slot 1 of `0x5ca1ab1e` and the `newStateRoot` are different because when the resulting state root is written at `0x5ca1ab1e` this modifies the final state root (`newStateRoot`).

`0x5ca1ab1e` and `BlockhashL2` at fork-etrog (fork-6)

In the zkEVM fork-etrog, similarly (but not exactly) as L1 Ethereum, more data related to the L2 block processing will be secured by the zkEVM.

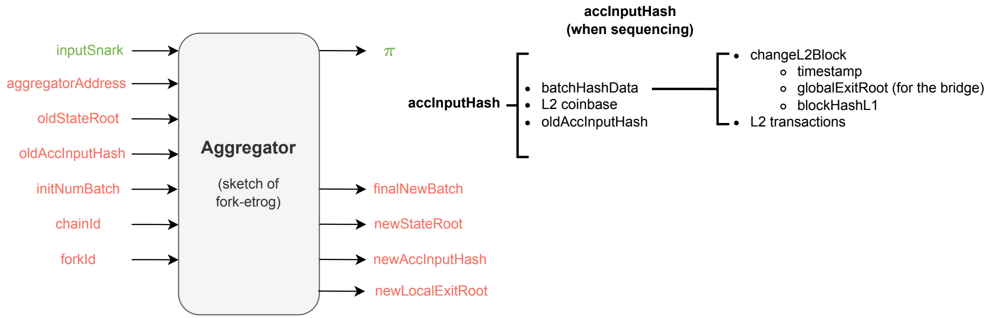
In particular, the L2 system smart contract under the address `0x5ca1ab1e` stores:

- At **slot 0**: the last block number processed.
- At **slots hash(1|blockNum)**: these slots are encoded as a Solidity mapping to store all the state roots indexed per block number.
- At **slot 2**: the timestamp of the last block processed.
- At **slot 3**: the root of a tree called `BlockInfoTree` which contains information about the execution of the last processed block.


5ca1ab1e
slot 0: LAST_BLOCK
slots hash(1|blockNum): STATE_ROOT
slot 2: TIMESTAMP
slot 3: BLOCK_INFO_ROOT

Sequencing and Proving up to fork-etrog (fork-6)

Here we can see an sketch (not exactly) of how the aggregator works in etrog.



BlockInfoTree: Transaction Data

- Data from the byte stream of L2 transactions:
  - `transactionHashL2` = LinearPoseidon(`txData`)
    with `txData` = (`nonce,gasPrice, gasLimit, to, value, data, from`).

    Note. Actually, the `from` field is computed from the signature with ecrecover.
  - `effectivePercentage` (zkEVM specific).

- Data from L2 transactions' execution:
  - `status`
  - `receiptData`
  - `cumulativeGasUsed`
  - `linearPoseidon(log_0_data)`
  - `linearPoseidon(log_1_data)`
  - ...
  - `linearPoseidon(log_N-1_data)`

BlockInfoTree: Common Block Data

- Data stored/updated from L2 state:
  - `Previous blockhashL2`.
  - `blockNumber`.
- Data from each L2 block (obtained from L1 as input for the proof):
  - `timestamp` .
  - `globalExitRoot` (for the bridge, explained later).
  - `blockHashL1`: L1 blockhash when the `globalExitRoot` parameter was recorded by the L1 contract `GlobalExitRoot` (in solidity this is done using `blockhash(block.number - 1)`).
- Data from the data of the sequence of batches (obtained from L1 as input for the proof):
  - `coinbase L2`.
- Data computed from the block execution:
  - `gasUsed`.
- Other parameters:
  - `gasLimit` (infinite but a single transaction is limited by 30M gas by the zkEVM processing).

Keys for the BlockInfoTree i

- We have to store all the previous block-related and transaction-related data in the BlockInfoTree.
- The BlockInfoTree is a read/write sparse Merkle Tree used as a key-value structure.
- The tree is built using the Poseidon hash function.
- The keys used to position each piece of data within the tree are also computed using the Poseidon hash function over some inputs.
- The Poseidon hash function that we use has the following signature:

$$\texttt{Poseidon}(\texttt{capacity}; \texttt{input1}, \texttt{input2}, \texttt{input3}, \texttt{input4})$$

  where `capacity` can be used as another input of the hash function.

Keys for the BlockInfoTree ii

- We use a parameter called `SMT_KEY_BLOCK_HEADER` that is fixed to the value 7 (`SMT_KEY_BLOCK_HEADER` = 7) to compute the key of block-related data.
- Additionally, we use a parameter called `INDEX_BLOCK_HEADER` to distinguish between the different block-related data:
    - `INDEX_BLOCK_HEADER = 0`: for the previous block hash.
    - `INDEX_BLOCK_HEADER = 1`: for the coinbase address.
    - `INDEX_BLOCK_HEADER = 2`: for the block number.
    - `INDEX_BLOCK_HEADER = 3`: for the gas limit.
    - `INDEX_BLOCK_HEADER = 4`: for the block timestamp.
    - `INDEX_BLOCK_HEADER = 5`: for the Global Exit Root.
    - `INDEX_BLOCK_HEADER = 6`: for the L1 block hash.
    - `INDEX_BLOCK_HEADER = 7`: for the gas used.
- Then, the associated keys are computed as follows:

$$\texttt{Poseidon}(0; \texttt{INDEX\_BLOCK\_HEADER}, 0, \texttt{SMT\_KEY\_BLOCK\_HEADER}, 0)$$

Keys for the BlockInfoTree iii

- The parameter `SMT_KEY_BLOCK_HEADER` takes the following values when computing the key transaction-related data:
  - `SMT_KEY_BLOCK_HEADER = 8`: for the transaction hash.
  - `SMT_KEY_BLOCK_HEADER = 9`: for the transaction status.
  - `SMT_KEY_BLOCK_HEADER = 10`: for the transaction cumulative gas used.
  - `SMT_KEY_BLOCK_HEADER = 11`: for the transaction logs.
  - `SMT_KEY_BLOCK_HEADER = 12`: for the transaction effective percentage.
- Then, the associated keys are computed as follows:

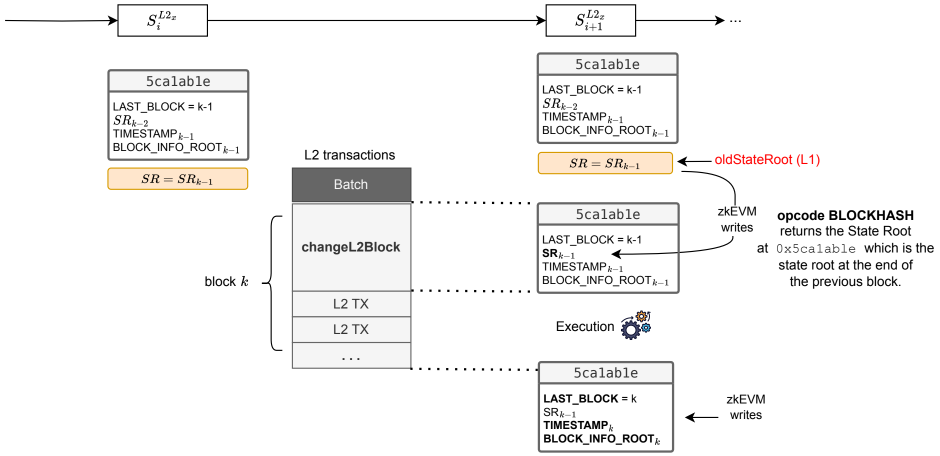
$$\texttt{Poseidon}(0; \texttt{txIndex}, 0, \texttt{SMT\_KEY\_BLOCK\_HEADER}, 0)$$

  where notice that we hash the transaction index within the block.
- Finally, since we can have multiple logs per transaction, the key for logs is computed as follows:

$$\texttt{Poseidon}(\texttt{logIndex}; \texttt{txIndex}, 0, \texttt{SMT\_KEY\_BLOCK\_HEADER}, 0).$$

  where notice that we hash the index of the log within the transaction.

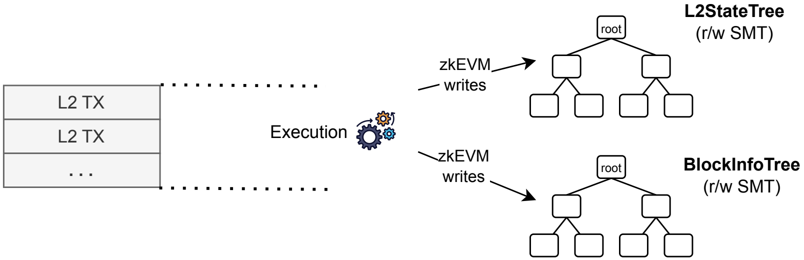
Processing L2 Blocks up to fork-etrog (fork-6) i



opcode BLOCKHASH returns the State Root at `0x5ca1ab1e` which is the state root at the end of the previous block.

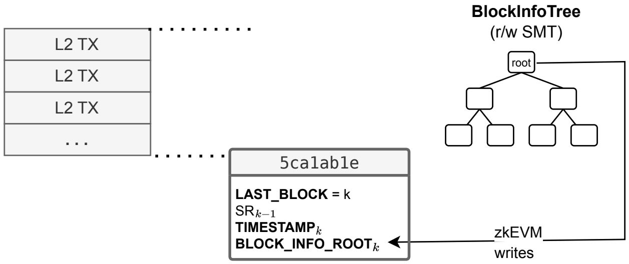
Processing L2 Blocks up to fork-etrog (fork-6) ii

- At the beginning of the processing of an L2 block, the zkEVM processing creates a **r/w sparse merkle tree** called **BlockInfoTree** to store information about the execution of the block.
- Each block uses a **new and empty BlockInfoTree**.
- As transactions of the block are being processed by the zkEVM, information related to the execution is stored in the **BlockInfoTree**.



Processing L2 Blocks up to fork-etrog (fork-6) iii

- The implementation of the **BlockInfoTree** is exactly the same as the r/w SMT used to store the L2 state tree.



- As usual, we use the root of the **BlockInfoTree** is used as the cryptographic summary of data contained by the tree.
- At the end of the block execution, the **BlockInfoTreeRoot** is stored at the system smart contract `0x5ca1ab1e`.

Processing L2 Blocks up to fork-etrog (fork-6) iv

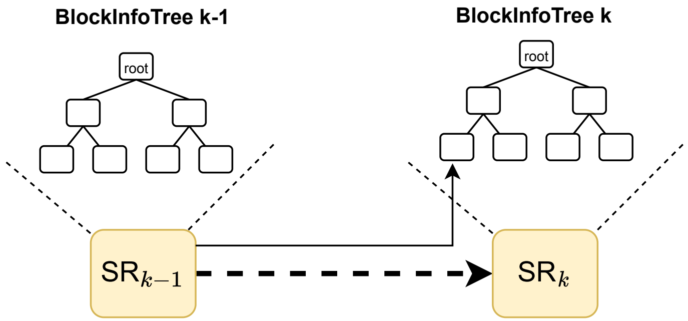
The values of the storage slots of `0x5ca1ab1e` are updated by the zkEVM processing as follows:

- The `LAST_BLOCK`, `TIMESTAMP` and `BLOCK_INFO_ROOT` are updated at the end of processing of the current block.
- However, the corresponding `STATE_ROOT` (used as `BLOCKHASHL2`) is not yet updated!
- The `STATE_ROOT` will be updated at the beginning of the next block when processing the `changeL2Block` transaction.
- In this way, the `STATE_ROOT` at the end of a block matches the `STATE_ROOT` stored in the mapping of `0x5ca1ab1e` which is stored at the beginning of the next block.

`BLOCKHASH`: L1 vs. zkEVM

- As it can be observed, the BLOCKHASH is computed differently than Ethereum.
- For example, in zkEVM, we include zkEVM-specific parameters like the `effectivePercentage`.
- Also, we build the hash using a different hash function and way of hashing.
- In particular, the transactions data is hashed with a linear poseidon, so we will have a `txHashL2` which differs from the L1.

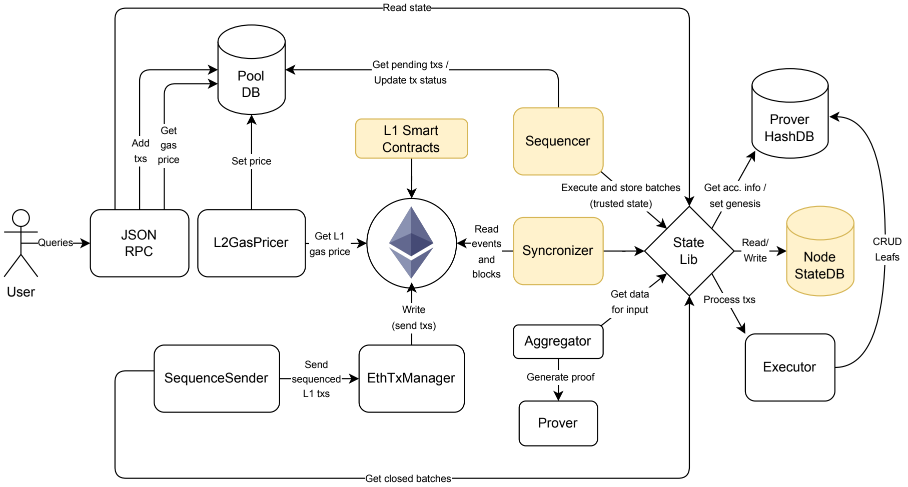
Linked BLOCKHASHes



The state roots are linked, so having a state root, e.g. SR_k we can prove any previous data since we have all the history, which can be proved using Merkle proofs.

Outline

Architecture of the zkEVM: Synchronizer



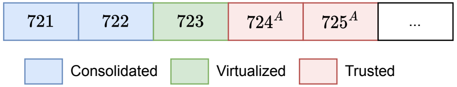
List of To Be Covered Concepts

- L2 reorganizations. ☐
- L1 reorganizations. ☐
- The synchronizer. ☐

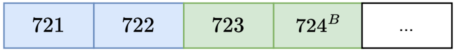
L2 Reorganizations i

Next, we show a situation in which there is a reorganization of L2 batches.

- Consider that there is a certain sequencer (let's call it sequencer A) that has closed the batch 724.
- So, the batch 724^A is in trusted state.



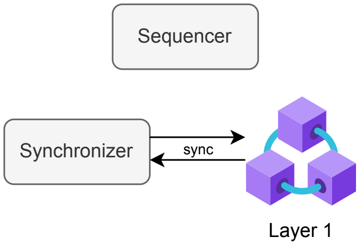
- However, let's consider that another sequencer (let's call it sequencer B) closes and sequences a different 724 batch:



- In this case, sequencer A will need to be aware of this situation and re-synchronize its state from 724^B.

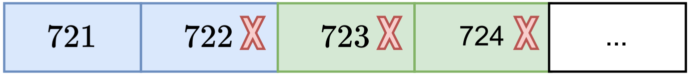
L2 Reorganizations ii

- More specifically, **sequencers** need to check the sequenced transactions present in L1 and re-synchronize their batch flow in case another sequencer virtualizes a different batch.
- In the zkEVM architecture, there is a component called **synchronizer** that checks the events produced in L1 when a batch is sequenced so that the sequencer can re-synchronize if needed.



L1 Reorganizations

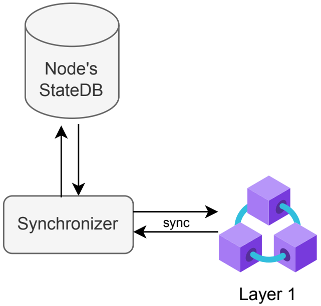
- L1 reorganizations happen if there is a reorg in Ethereum itself.
- In general, L1 reorganizations should never happen.
- Moreover, these reorganizations are far more critical, since it might be the case that we have to re-synchronize already virtualized and/or consolidated batches.
- The **synchronizer** is also in charge of detecting these situations and inform the **sequencer** so that the reorg can be performed.



Synchronizer

Synchronizer

In general, the **synchronizer** detects and records in the node's **StateDB** any relevant event from L1 (not only reorgs).



List of Covered Concepts

- L2 reorganizations. ☑
- L1 reorganizations. ☑
- The synchronizer. ☑