The zkEVM Architecture

Part III: Communication Between Layers

Polygon zkEVM & Universitat Politècnica de Catalunya (UPC)

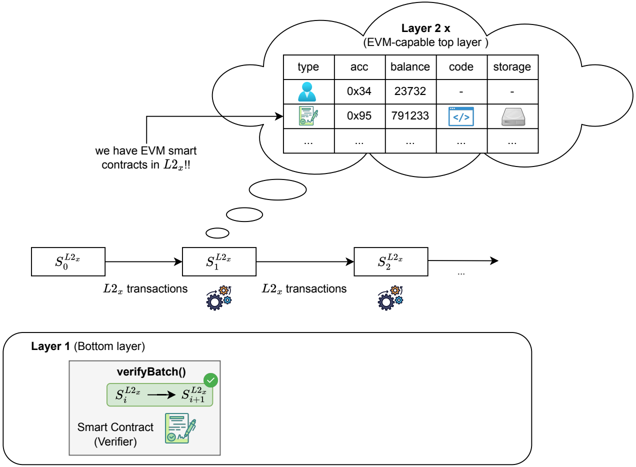
Marc Guzman-Albiol <marc.guzman.albiol@upc.edu>
Jose Luis Muñoz-Tapia <jose.luis.munoz@upc.edu>

Version 1.2

December 12, 2023

Outline

L2 zkEVM



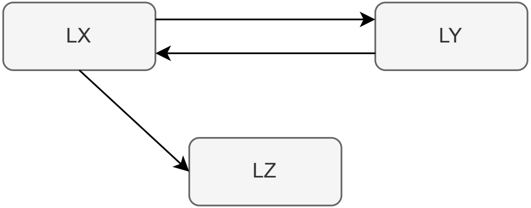
Layer 2 x
(EVM-capable top layer)
type
acc
balance
code
storage
0x34
23732
-
-
0x95
791233
...
we have EVM smart contracts in $L2_x$!!
$S_0^{L2_x}$
$S_1^{L2_x}$
$S_2^{L2_x}$
$L2_x$ transactions
$L2_x$ transactions
...
Layer 1 (Bottom layer)
verifyBatch()
$S_i^{L2_x} \longrightarrow S_{i+1}^{L2_x}$
Smart Contract (Verifier)

The Bridge

Bridge

The bridge is a subsystem of the zkEVM that is composed of several components and its main purpose is to enable **exchanges between different layers**.

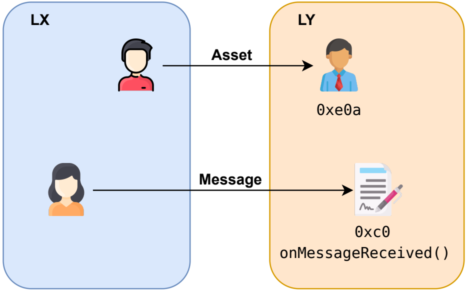
- We will start by defining exchanges between L1 and an L2 but our intention is to be general, that is, to enable exchanges between multiple layers LX and LY.
- This is why we call this subsystem the **LXLY bridge**.
- For the explanations, we will use three layers denoted as LX, LY and LZ.



Exchanges: Assets and Messages

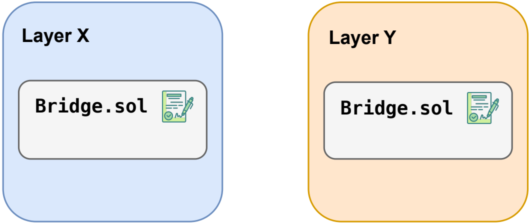
We enable the exchanges of:

a) **Assets:** Ether or Tokens to accounts in the destination layer.
b) **Messages:** The execution of a function `onMessageReceived` of some contract.
   This is what we call the **messaging** mechanism of the bridge (messages can transfer Ether too).



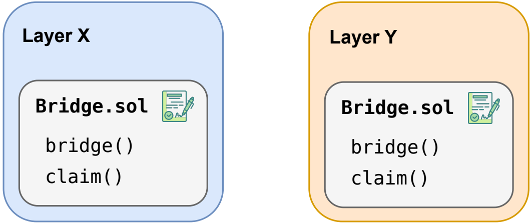
LXLY Bridge and Smart Contracts

- The core logic of the LXLY bridge is implemented in smart contracts.
- In particular, the main contract is a contract called `Bridge.sol` that is deployed in any layer in which we want exchanging enabled.
- One of our goals is that the `Bridge.sol` smart contract **is exactly the same in all layers**.



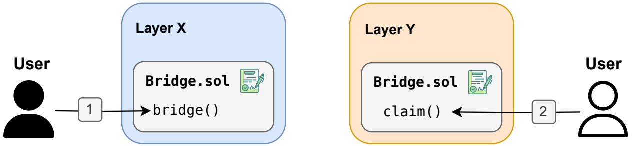
LXLY Exchanges: Bridge-Claim Model i

The LXLY bridge follows a **bridge-claim model**:



As it can be observed, each bridge smart contract has the methods `bridge()` and `claim()`.

LXLY Exchanges: Bridge-Claim Model ii



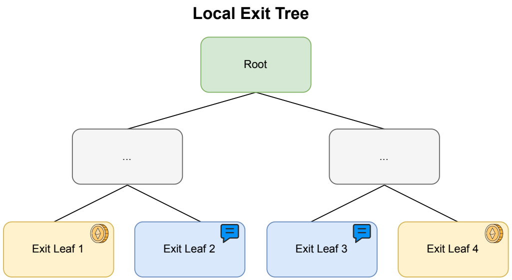
1. In the origin layer (e.g. LX), the user sends a transaction to the **`bridge()`** function providing the destination network (e.g. LY). Transactions to the bridge function are also known as ”deposits”.
2. In the destination layer (LY), the user sends a transaction to the **`claim()`** function providing the origin network (e.g. LX).

In the `Bridge.sol` smart contracts, we need a compact way of storing the information of calls to the bridge function (that we also call these data ”exits” or ”outgoing transmissions”).

Outline

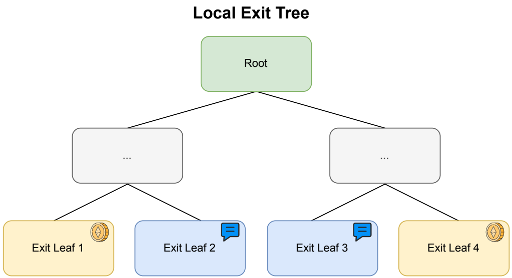
Local Exit Tree

- In each layer, the bridge contract builds an **append-only Merkle tree** with all the exits, i.e. each call to the `bridge()` function.
- The Merkle tree with all the exits of a layer is called its **Local Exit Tree** and, its root is called `Local Exit Root` (LER).
- Each leaf of a Local Exit Tree of a layer LX stores a single exit.



Remark. An exit tree is a different object and has a different structure than the tree that stores the L2 state.

Leaves of Exit Trees i



Each leaf contains:

```
uint8 leafType,
uint32 originNetwork,
address originAddress,
uint32 destinationNetwork,
address destinationAddress,
uint256 amount,
bytes32 metadataHash
```

Leaves of Exit Trees ii

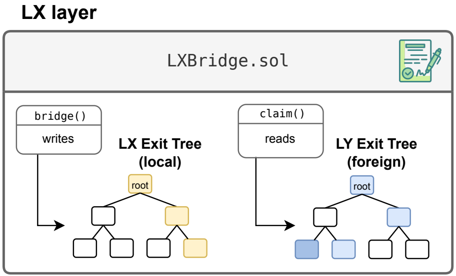
- `leafType` used to identify whether the leaf is an asset or a message:
  - Asset: value is 0.
  - Message: value is 1.
- `originNetwork`: the identifier (`chainId`) of origin layer of the exchange.
- `originAddress`: if it is an asset exchange, it is the address of the token contract.

  If it is an message exchange, it is the source address of the bridge call.
- `destinationNetwork`: the identifier of the destination layer (`chainId`) of the exchange.
- `destinationAddress`: is the account receiving the asset or the address of the smart contract if it is a message exchange.
- `leafAmount`: amount of asset exchanged (Ether or Tokens).
- `bytes32 metadataHash`: the hash of the metadata.
  - Asset: the metadata is the name, symbol and decimals of the token.
  - Message: the metadata is the `calldata` for calling the `onMessageReceived()` function.

Read and Write Exit Trees

In each layer, the corresponding bridge smart contract needs to:

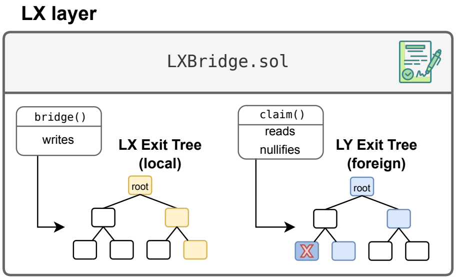
- **Write**: write in its Local Exit Tree each new exit resulting from a call to its `bridge()` function.
- **Read**: read the Exit Trees of other layers to process calls to `claim()`.



Read and Nullify

In each layer, the corresponding bridge smart contract needs to additionally **nullify** the claim:

- Each claim must be **locally nullified in the bridge contract to avoid double claimings**.
- The nullify process to avoid claiming transactions that have already been processed uses an efficient mapping known as `claimedBitMap` (we will explain how this bitmap works later in more detail).



Outline

The idea is to create an append-only sparse tree whose successive roots can be computed with a minimal amount of persistent data.

- It turns out that to append new data elements to the tree we are going to just need to store:
  1. An array of the size of the tree depth (denoted as `branch`).
  2. The last appended element's index (denoted as `lastElemIndex`).
- The depth of the tree (maximum capacity) will be known a priori.
- As a result, we will not need to use markers for branches and leaves.
- Furthermore, the tree will be balanced.

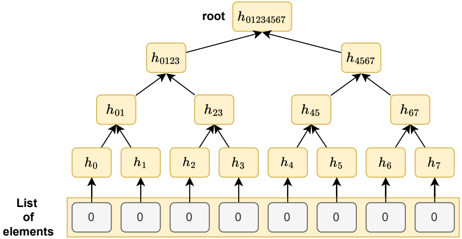
Empty Append-only SMT i

For our append-only tree we are going to use 0s as default value for empty leaves.

When the incremental Merkle tree is empty, we have that:

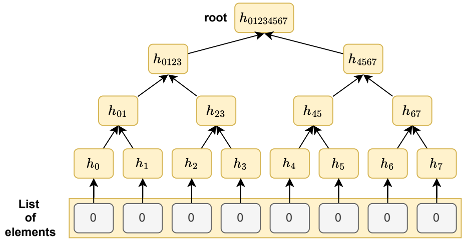
$$
\begin{aligned}
h_i &= 0 \\
h_{j,k} &= h(0|0) = h^{(00)} \\
h_{m,n,\ell} &= h(h^{(00)}|h^{(00)}) = h^{(0000)} \\
&\dots
\end{aligned}
$$

- So, we just need to compute a different hash value per level.
- Let's consider as a toy example a small incremental Merkle tree of a maximum capacity of 8 leaves ($d = 3$).



Empty Append-only SMT

Empty Append-only SMT ii



Empty Append-only SMT

$$\texttt{lastElemIndex} = 0 \quad \texttt{branch} = [0, 0, 0]$$

$$\texttt{root} = h^{(00000000)}$$

$$h^{(00000000)} = h(h^{(0000)}|h^{(0000)})$$

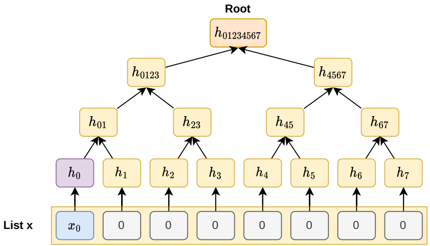
$$h^{(0000)} = h(h^{(00)}|h^{(00)})$$

$$h^{(00)} = h(0|0)$$

- Note that we can compute the root of the empty tree from the zero hash values.
- Note also that the hash of zero nodes is uniquely determined by the height of the subtree under the node.

Append the First Element

Let's suppose that we want to add an element x_0:



$$\texttt{lastElemIndex} = 1 \quad \texttt{branch} = [h_0, 0, 0]$$

$$\texttt{root} = h_{01234567}$$

$$h_{01234567} = h(h_{0123}|h^{(0000)})$$

$$h_{0123} = h(h_{01}|h^{(00)})$$

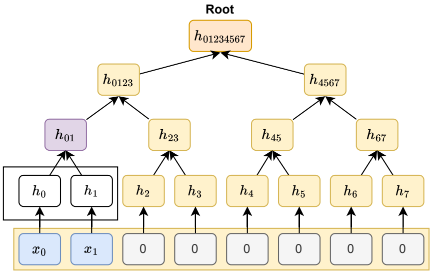
$$h_{01} = h(h_0|0)$$

$$h_0 = h(x_0)$$

- Note that we "can" (lazy) compute the root of the current tree with the **branch** and **lastElemIndex**.
- Just need to write **branch**$[0] = h_0 = h(x_0)$.

Append an Second Element

Let us now add another element x_1 into the tree:



$$\texttt{lastElemIndex} = 2 \quad \texttt{branch} = [h_0, h_{01}, 0]$$

$$\texttt{root} = h_{01234567}$$

$$h_{01234567} = h(h_{0123}|h^{(0000)})$$

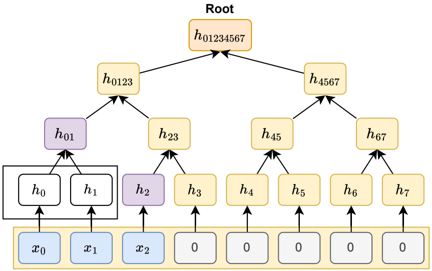
$$h_{0123} = h(h_{01}|h^{(00)})$$

$$h_{01} = h(h_0|h(x_1))$$

- Note that we can compute the root of the current tree by writing $\texttt{branch}[1] = h_{01} = h(h_0|h(x_1))$ in the $\texttt{branch}$.
- From now on, h_0 and h_1 are not needed any more for updating the root since they are integrated in h_{01}.

Append a Third Element

Let us now add another element x_2 into the tree:



$$\texttt{lastElemIndex} = 3 \quad \texttt{branch} = [h_2, h_{01}, 0]$$

$$\texttt{root} = h_{01234567}$$

$$h_{01234567} = h(h_{0123}|h^{(0000)})$$

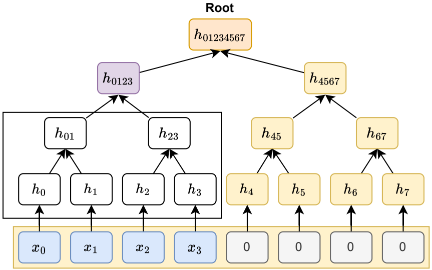
$$h_{0123} = h(h_{01}|h_{23})$$

$$h_{23} = h(h_2|0)$$

$$h_2 = h(x_2)$$

Append a Fourth Element

Let us now add another element x_3 into the tree:

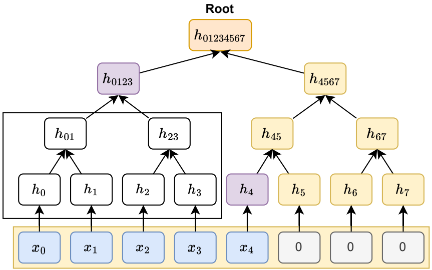


$$\texttt{lastElemIndex} = 4 \quad \texttt{branch} = [h_2, h_{01}, h_{0123}]$$
$$\texttt{root} = h_{01234567}$$
$$h_{01234567} = h(h_{0123}|h^{(0000)})$$
$$h_{0123} = h(h_{01}|h_{23})$$
$$h_{23} = h(h_2|h_3)$$
$$h_3 = h(x_3)$$

From now on, h_0, h_1, h_2, h_3, h_{01} and h_{23} are not needed any more for updating the root since they are integrated in h_{0123}.

Append a Fifth Element

Let us now add another element x_4 into the tree:



$$\texttt{lastElemIndex} = 5 \quad \texttt{branch} = [h_4, h_{01}, h_{0123}]$$

$$\texttt{root} = h_{01234567}$$

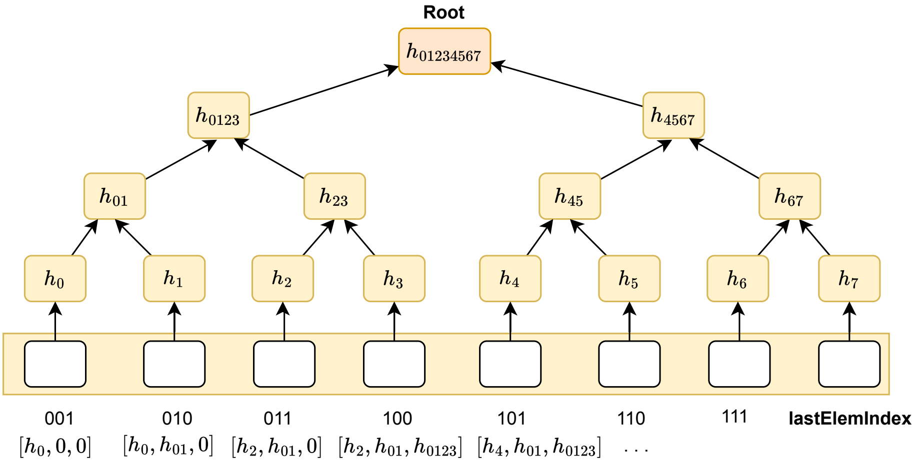
$$h_{01234567} = h(h_{0123}|h_{4567})$$

$$h_{4567} = h(h_{45}|h^{(00)})$$

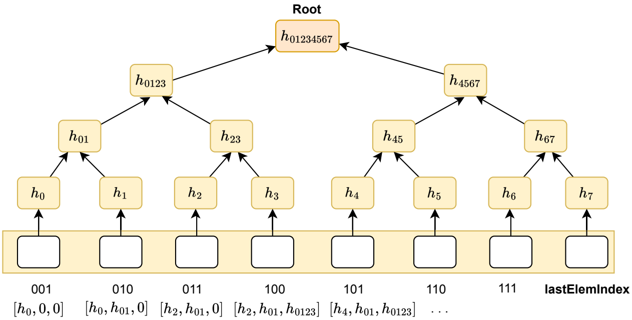
$$h_{45} = h(h_4|0)$$

$$h_4 = h(x_4)$$

The Written Position in the **branch** i

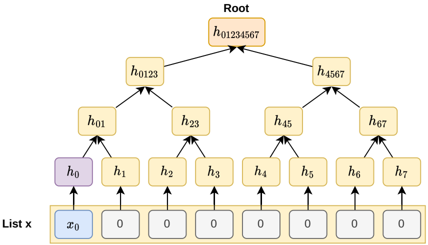


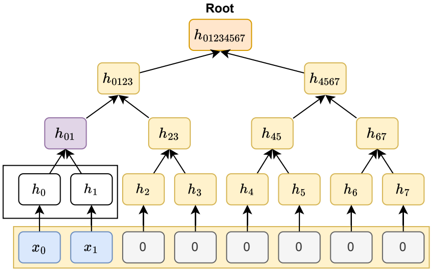
The Written Position in the `branch` ii



In general, the position of the **branch** that we have to update corresponds to the position of the **less significant bit to** 1 **of the binary representation of `lastElemIndex`**.

The Written Position in the **branch** iii

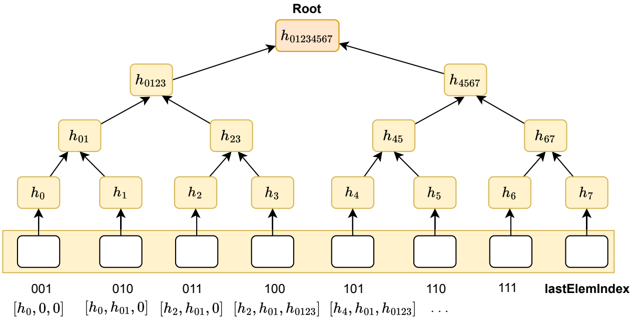




$\texttt{lastElemIndex} = 1\ (\texttt{0b001})$ $\quad \texttt{branch} = [h_0, 0, 0]$

$\texttt{lastElemIndex} = 2\ (\texttt{0b010})$ $\quad \texttt{branch} = [h_0, h_{01}, 0]$

Writing the **branch** i

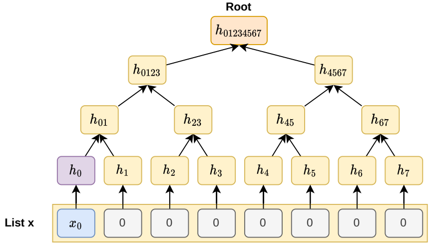


The following pseudo-code computes the value and position to write in the **branch** array:

```
currentHash = dataHash
index = 0

while read bit of lastElemIndex next lsb {
  if(bit == 0) {
    currentHash = h(branch[index]|currentHash)
  } else { // bit == 1
    branch[index]= currentHash
    return
  }
  index++
}
```

Writing the **branch** ii

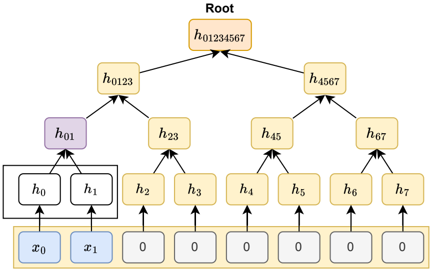


lastElemIndex $= 1$ (0b001) branch $= [h_0, 0, 0]$

```
currentHash = dataHash
index = 0

while read bit of lastElemIndex next lsb {
  if(bit == 0) {
    currentHash = h(branch[index]|currentHash)
  } else { // bit == 1
    branch[index]= currentHash
    return
  }
  index++
}
```

Writing the branch iii

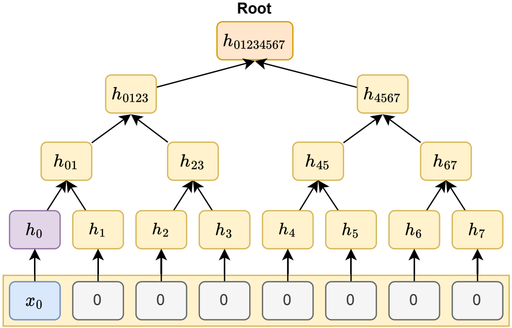


lastElemIndex $= 2$ (0b010)   branch $= [h_0, h_{01}, 0]$

```
currentHash = dataHash
index = 0

while read bit of lastElemIndex next lsb {
  if(bit == 0) {
    currentHash = h(branch[index]|currentHash)
  } else { // bit == 1
    branch[index]= currentHash
    return
  }
  index++
}
```

Computing the Root i



lastElemIndex = 1 (0b001)
branch = $[h_0, 0, 0]$

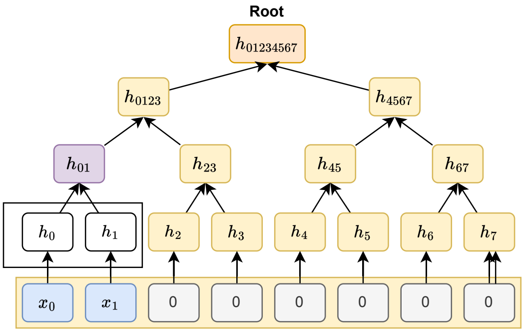
index = 2 bit = 0
currentHash = h(currentHash | currentZeroHash) = root

index = 1 bit = 0
currentHash = h(currentHash | currentZeroHash) = h_{0123}
currentZeroHash = h(currentZeroHash | currentZeroHash) = $h^{(0000)}$

index = 0 bit = 1
currentHash = h(branch[index] | currentHash) = h_{01}
currentZeroHash = h(currentZeroHash | currentZeroHash) = $h^{(00)}$

currentZeroHash = 0
currentHash = currentZeroHash

Computing the Root ii



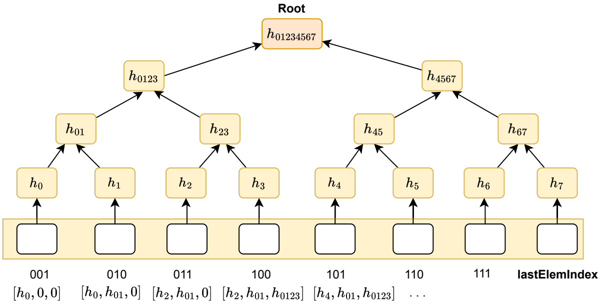
lastElemIndex = 2 (0b010)
branch = $[h_0, h_{01}, 0]$

index = 2 bit = 0
currentHash = h(currentHash | currentZeroHash) = root

index = 1 bit = 1
currentHash = h(branch[index] | currentHash) = h_{0123}
currentZeroHash = h(currentZeroHash | currentZeroHash) = h $^{(0000)}$

index = 0 bit = 0
currentHash = h(currentHash | currentZeroHash) = h_{23}
currentZeroHash = h(currentZeroHash | currentZeroHash) = h $^{(00)}$

currentZeroHash = 0
currentHash = currentZeroHash



The following pseudo-code computes the root:

```
currentZeroHash = 0
currentHash = currentZeroHash
index = 0

while read bit of lastElemIndex next lsb {
  if(bit == 0) {
    currentHash = h(currentHash | currentZeroHash)
  } else { // bit == 1
    currentHash = h(branch[index] | currentHash)
  }
  currentZeroHash = h(currentZeroHash | currentZeroHash)
  index++
}
```

Some Properties of the Append-only SMT

- Note that we built a key-value tree that is sparse, binary and balanced.
- We also applied partial tree construction and we did lazy evaluation in the computation of the root.
- Insertions only need to write one position in the `branch` array.
- Finally, we could revoke” or ”nullify” elements if we use an extra structure to denote which elements of the append-only tree are no longer valid.

Outline

Smart Contract for the Exit Tree

The smart contract to manage the exits can be found at:

https://github.com/0xPolygonHermez/zkevm-contracts/blob/main/contracts/lib/DepositContract.sol

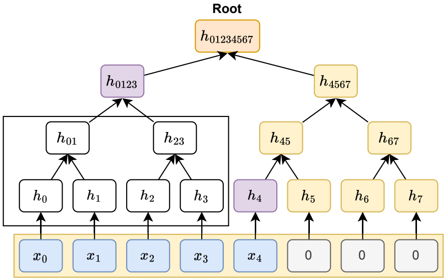
The contract is based on the implementation of the Eth 2.0 deposit contract.

The contract has four functions:

- `function getLeafValue(...)` which computes the hash of a leaf.
- `_deposit(bytes32 leafHash)` which adds a new leaf to the tree
- `getDepositRoot()` which computes the Merkle root.
- `function verifyMerkleProof(...)` which verifies a Merkle proof.

Obtaining Merkle Proofs of Previous Deposits

- Notice that since the branch is updated, the smart contract is not able to provide the Merkle proof for old deposits.
- In the example, the Merkle proof for x_0 is the tuple (h_1, h_{23}, h_{4567}) which is not currently stored on the smart contract.



$lastElem = 5 \quad branch = [h_4, h_{01}, h_{0123}]$

The Claim Service

- If we want to simplify the process of obtaining a Merkle proof for users, we have to store all the Merkle tree nodes and provide a service that answers with the appropriate Merkle proofs.
- This service in the zkEVM is called the **claim service**.
- The implementation is called claim service manager and it is available at:

  https://github.com/0xPolygonHermez/zkevm-bridge-service/tree/develop/claimtxman

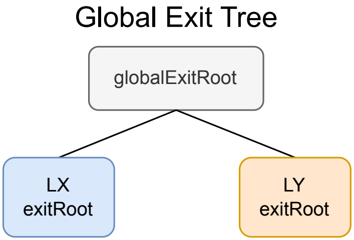
Outline

The Global Exit Tree

So, while processing a `claim()` at a certain layer, this layer might need to read an exit leaf stored in the Exit Tree of another layer.

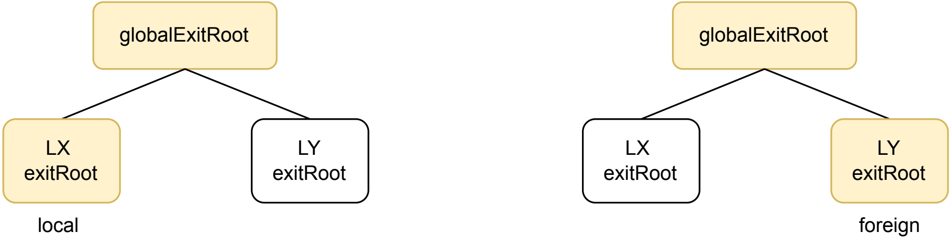
For enabling this, we build a Merkle tree that includes all the exit trees of all the layers.

- In particular, the local exit roots are used as leaves to build the **Global Exit Tree**.
- The root of the Global Exit Tree will be called `globalExitRoot` and it is a cryptographic digest of all the Local Exit Roots it contains.
- With the `globalExitRoot` and the appropriate **Merkle proofs**, a bridge contract can securely **read a leaf from any exit tree**.



Updating the `globalExitRoot`

- From the point of view of a layer, the Global Exit Tree can change because:
  - The local exit tree of the layer changes.
  - The local exit tree of another layer changes.



LX: Local Exit Root Update

LX: Foreign Exit Root Update

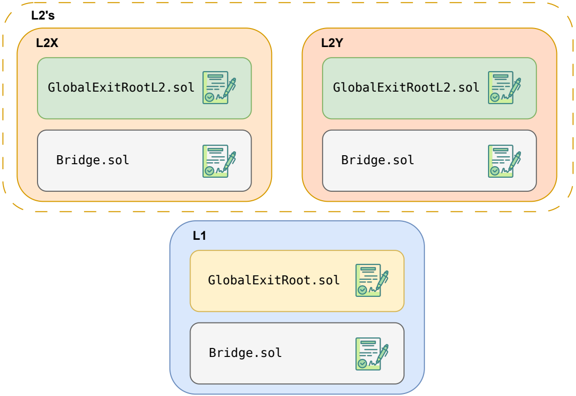
The `globalExitRoot` Update is Layer-specific i

- Each Bridge smart contract does the update of its local exit root with the same logic when there are calls to the `bridge()` function.
- However, as we will see next, the `globalExitRoot` update is performed differently in the bottom layer (L1) than in a top layer (L2) when there is an update of a foreign exit root.
- In other words, the logic to update the global exit root is layer-specific.
- But, recall that we want to deploy the same bridge smart contract in all the layers.
- So, to achieve this, the solution is to deploy the logic for updating the `globalExitRoot` in to a separate smart contract in each layer.

The `globalExitRoot` Update is Layer-specific ii

The contract that manages the update of the `globalExitRoot` in the bottom layer (L1) is called `GlobalExitRoot.sol`.

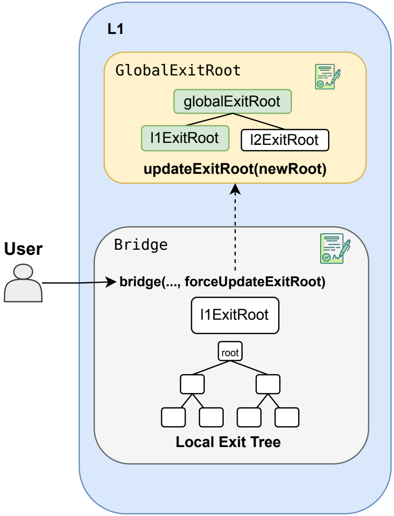
The contract that manages the update of the `globalExitRoot` in a top layer is called `GlobalExitRootL2.sol`.



Outline

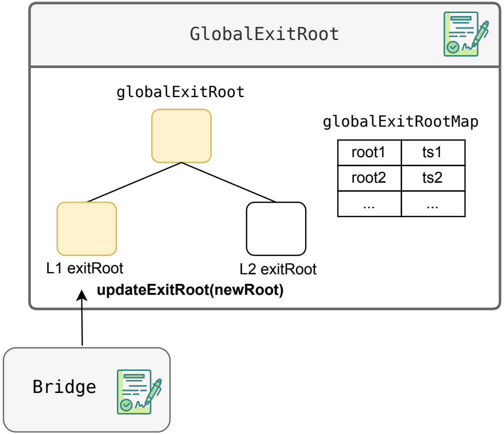
Updating the globalExitRoot in L1: Local Update

- When a user sends a transaction with a call to `bridge()` (i.e. if someone makes a deposit) the corresponding local exit root (in this case L1) is modified.
- The local exit root is managed by the Bridge smart contract.
- The execution of the bridge (deposit) **may update** the `globalExitRoot`, which is stored in the `GlobalExitRoot` smart contract, by calling the `updateExitRoot()` function.
- If the bool parameter `forceUpdateExitRoot` is set as true when calling the `bridge()` function, the bridge smart contract calls `updateExitRoot()` in the `GlobalExitRoot` contract to perform the update.



globalExitRootMap

`GlobalExitRoot.sol` contains a mapping that associates a `globalExitRoot` with the update timestamp, providing a mechanism to keep track of each root update.



About the `forceUpdateExitRoot` flag

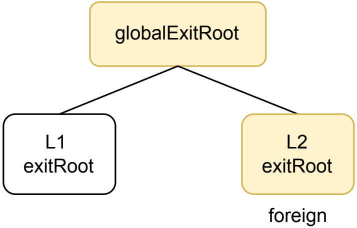
- The bool `forceUpdateExitRoot` can be set as:
    - `true`: the transaction costs around 120k gas.
    - `false`: the transaction costs around 40k gas (about 30% of the cost with the flag activated).
- The idea of the `updateGlobalExitRoot()` function in `Bridge.sol` is to be able to generate a sponsored `globalExitRoot` update service.

More Ways of Doing the `globalExitRoot` Local Update

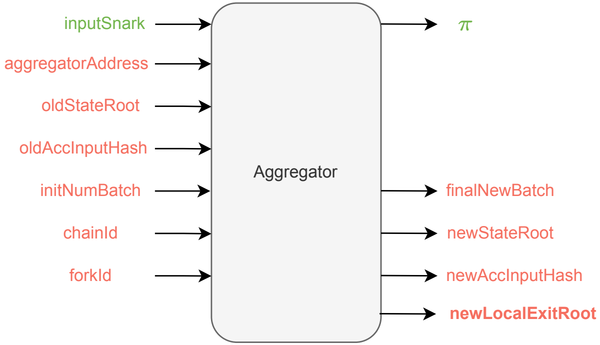
- Let's consider that happens if no user calls the `bridge()` function with `forceUpdateExitRoot` flag equal to `true`.
- Then, the question is who updates the `globalExitRoot` in the L1 `GlobalExitRoot` smart contract and when.
- In the zkEVM architecture, two actors can perform that job:
  a) `Anyone` can send a transaction to the `updateExitRoot()` function of the L1 `Bridge` smart contract.
  b) The `sequencer`, when calling `sequenceBatches()` function of the `ZkEVM` smart contract at the end calls the `updateExitRoot()` function of the `Bridge` contract.

Updating the `globalExitRoot` in L1: Remote Update

- Suppose there is an outgoing transmission from Layer 2, that is, an L2 transaction calling the `bridge()` function of the `Bridge` smart contract in L2.
- This L2 transaction changes the state of the L2 exit tree.
- **Q**: How does L1 realizes that the L2 `exitRoot` has changed?

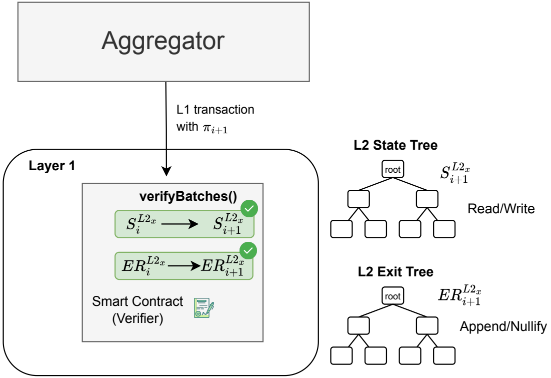


To make the state of the `L2 exitRoot` available to L1, we add a new parameter in the proof that contains the new `L2 exitRoot` that results after processing the batch.



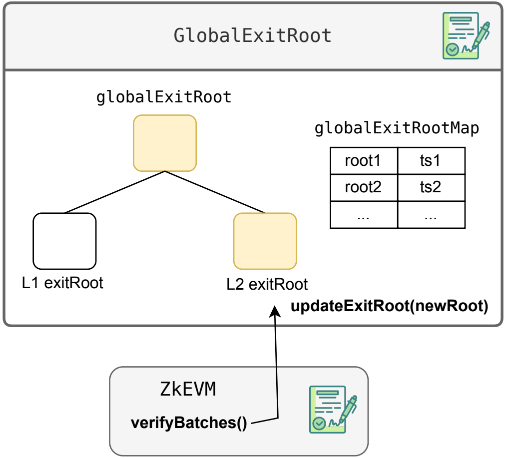
So, our new proof now assures that:

1. The processing of the L2 transactions is correct and so it is the new L2 state root (`newStateRoot`).
2. The new `L2 exitRoot` (`newLocalExitRoot`) is also correct.



Processing the Proofs in L1

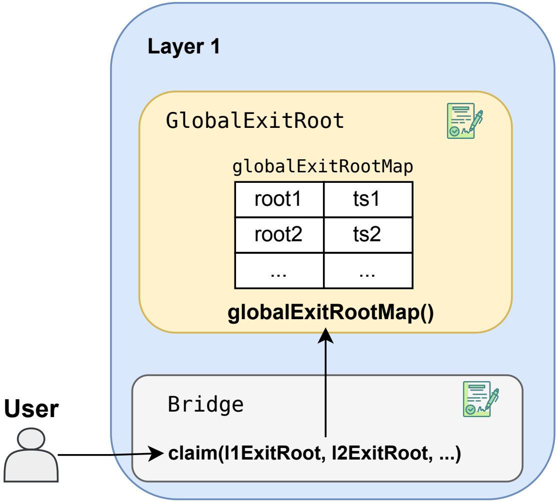
- The `verifyBatches()` function of `ZkEVM.sol` is who verifies the proofs.
- If the proof is correct, at the end will call the `updateExitRoot()` of the `GlobalExitRoot.sol` smart contract to perform the **`globalExitRoot`** update (providing the exit three of the foreign layer).
- As previously mentioned, all the `globalExitRoot`s are registered in a mapping in the `GlobalExitRoot.sol` smart contract.
- We can obtain the latest root calling `getLastGlobalExitRoot()` at `GlobalExitRoot.sol`.



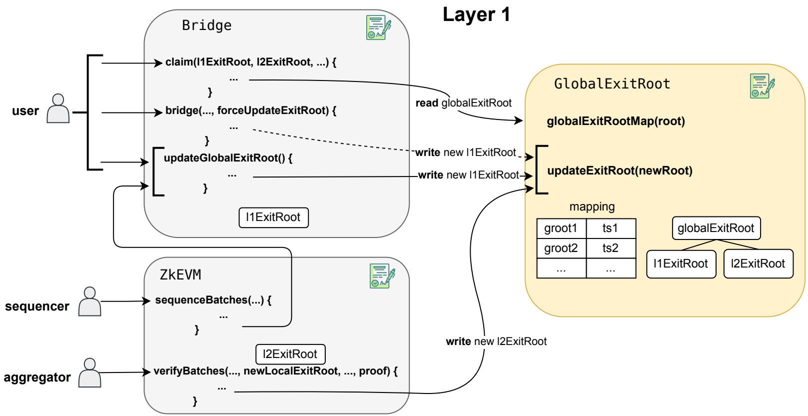
https://github.com/0xPolygonHermez/zkevm-contracts/blob/main/contracts/PolygonZkEVMGlobalExitRoot.sol

Reading the **globalExitRoot** in L1

When a user sends a transaction with a call to `claim()`, that is to say, when someone makes a withdraw, the `Bridge` smart contract needs to read a valid `globalExitRoot` to perform the claim.



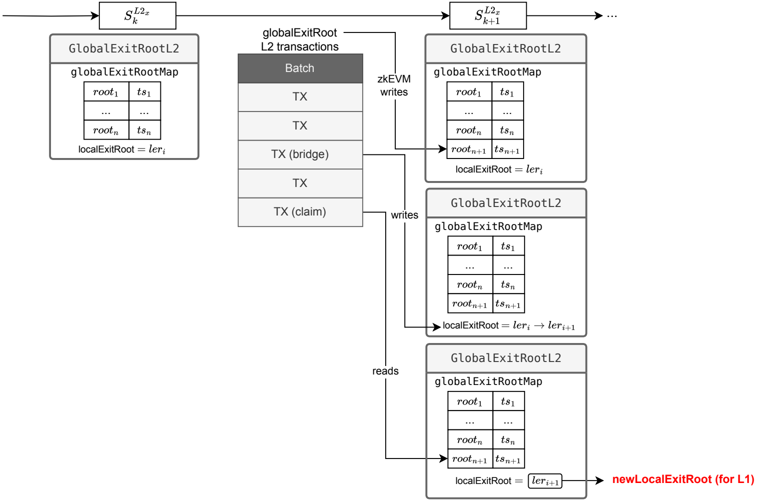
Summary of the **globalExitRoot** Management in L1



Outline

- The L2 also needs to be aware of changes in exit trees, that is, we have to manage local and remote updates of the exit trees in L2.
- In summary, this is done as follows:
    - The L2 local `exitRoot` is updated for each successful deposit (L2 transaction to `bridge()`).
    - The `globalExitRoot` is updated only **at the beginning of the processing of the L2 batch**.
    - Each batch, apart from the L2 transactions, includes a `globalExitRoot`.
    - The zkEVM processing of the batch inserts this `globalExitRoot` in the `globalExitRootMap` of the `GlobalExitRootL2` contract instance at L2.
    - The new `globalExitRoot` or any other previous root available in the `globalExitRootMap` can be used for doing the L2 claim transactions.

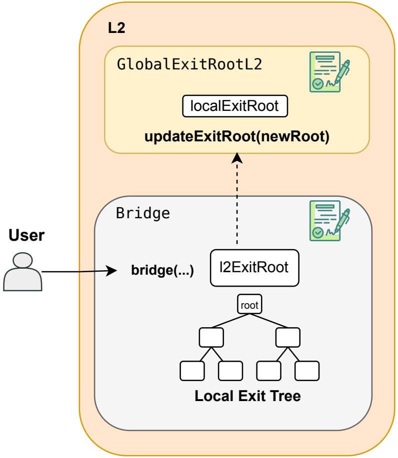
Overview of Exit Trees Updates in L2 ii



Performing bridges (deposits) in L2

Processing transactions that call `bridge()` in L2 is identical to what we do in L1:

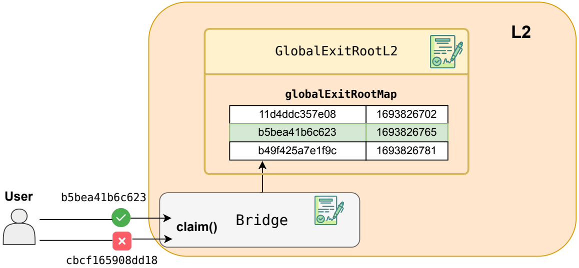
- When a user sends an L2 transaction with a call to `bridge()`, that is, when someone makes a deposit, the corresponding local exit root (in this case L2) is modified.
- If the bool parameter `forceUpdateExitRoot` is set as true when calling the `bridge()` function, the `Bridge` smart contract calls `updateExitRoot()` in the `GlobalExitRootL2` contract to perform the update.



Performing claims (withdraws) in L2

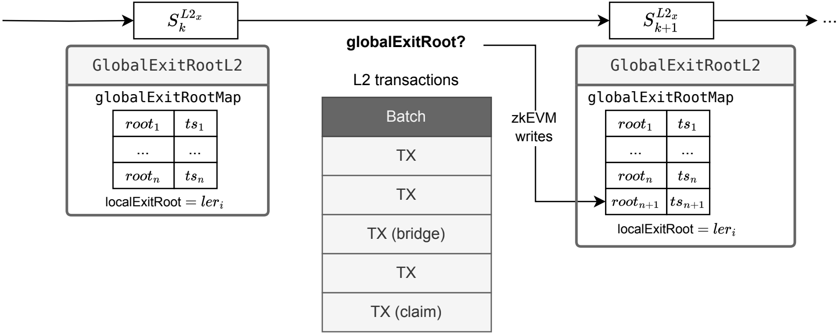
To do a claim in L2, we weed to access a `globalExitRoot` that includes the data of our deposit in the origin network (L1).

- To make updated `globalExitRoot`s available to L2, a `globalExitRoot` is provided with each batch.
- At the beginning of the ROM processing of a L2 batch, the executor inserts the `globalExitRoot` (and the corresponding timestamp) received as an input in the `globalExitRootMap` of `GlobalExitRootL2` contract.
- Finally, the `Bridge` smart contract in L2 can check if the deposit exists if the user provides a valid Merkle proof against a global exit root that is registered in the mapping of the `GlobalExitRootL2` contract.



Who Decides the `globalExitRoot` for L2? i

Who decides the `globalExitRoot` that goes with the batch?



Who Decides the `globalExitRoot` for L2? ii

- Note that if a `globalExitRoot` is registered or not influences the result of the batch execution.
- In more detail, depending on the available `globalExitRoot`s, some claims will be valid and some won't.
- Recall that sequencer does a batch pre-execution to check if the batch fits in the available resources so, the sequencer must deterministically know the result of the execution.
- Therefore, the sequencer is the actor that must associate a `globalExitRoot` for each batch.
- When sequencing the batch, the `ZkEVM` contract checks that the `globalExitRoot` provided by the sequencer exists in the mapping of the L1 `GlobalExitRoot` contract.
- Notice that the sequencer decides the `globalExitRoot` but, the aggregator also needs to read this value to generate the corresponding proof.

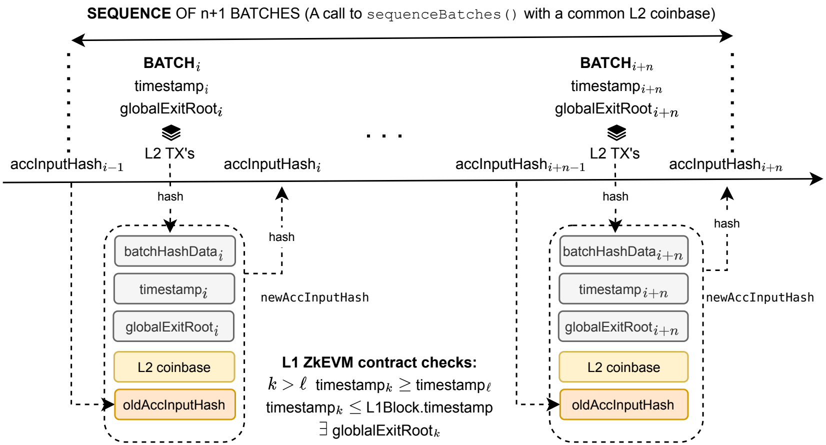
How is the `globalExitRoot` for the Batch Available to the Aggregator?

- The aggregator needs to use the same `globalExitRoot` decided by the sequencer to build the proof of a batch.
- To achieve this, the `globalExitRoot` is passed by the sequencer as part of the batch data when the batch is sequenced:

```
struct BatchData {
  bytes transactions;
  bytes32 globalExitRoot;
  uint64 timestamp;
}
```

- Then, the smart contract includes this parameter as part of the hashed input data to build the cryptographic pointer for the batch (`accInputHash`).

Sequences of Batches i



Sequences of Batches ii

- The **zkEVM smart contract checks** that:

  a) The **timestamps** of batches are correct:

  - The timestamps of a batch is set by the sequencer when the batch is started being filled with transactions.
  - Then, for batches in the sequence with $k > \ell$, then:

$$timestamp_k > timestamp_\ell.$$

  - All timestamps in the sequence are smaller than the timestamp of the L1 sequencing transaction:

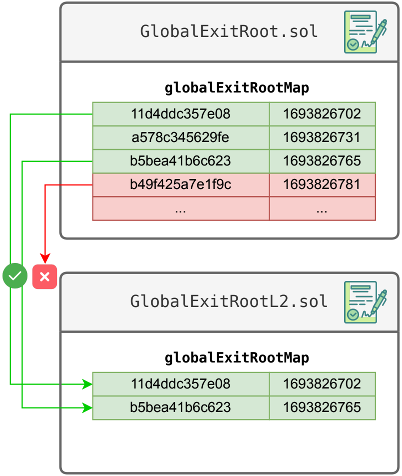
$$timestamp_k < block.timestamp.$$

  - Recall that in fork-dragonfruit (fork-5) all the blocks (transactions) within a block share the same timestamp.

  b) The **globalExitRoot** provided for each batch exists.

Censoring the **globalExitRoot** Progress (Avoids Withdraws)

- Notice that a sequencer may refuse to make the `globalExitRoot` progress.
- The situation is that there are new `globalExitRoot`s available in the `GlobalExitRoot` contract at L1, but the sequencer refuses to use them for sequenced batches.
- Notice that a user can do a claim using any already registered global root.
- But the withdraw of a deposit of a user can be effectively censored if the sequencer, when sequencing the batches, does not include an updated `globalExitRoot` that includes the data of the deposit.



Countermeasure for the Censoring of the `globalExitRoot`

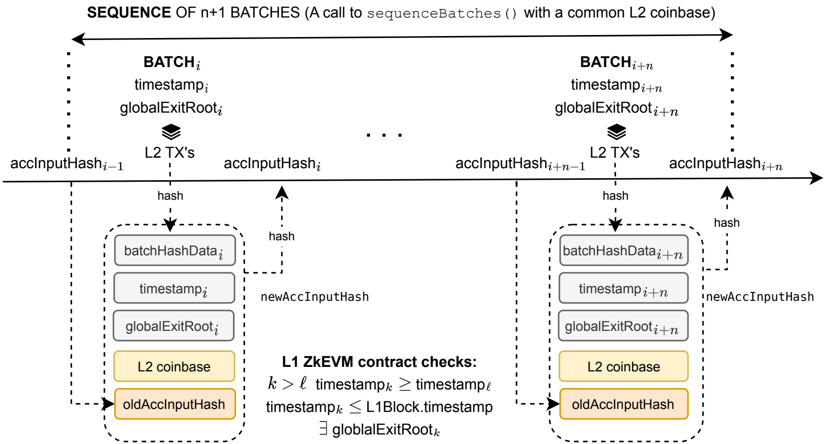
- To fix the problem of the censorship of the `globalExitRoot` we can use the same mechanism as we use to avoid the censorship of L2 transactions.
- This mechanism is the so called ”forced batches”.
- In a forced batch you can include the transactions that you like and also the `globalExitRoot` that you like:

```
struct ForcedBatchData {
        bytes transactions;
        bytes32 globalExitRoot;
        uint64 minForcedTimestamp;
    }
```

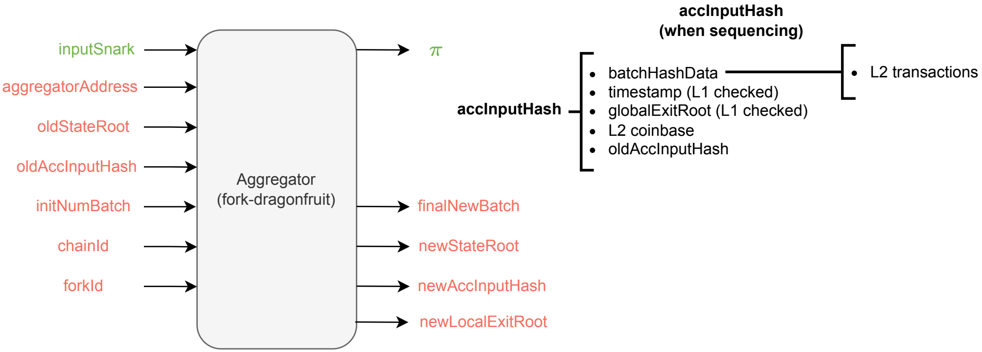
- In this way, you can enable your deposits for their corresponding withdraws.

Outline

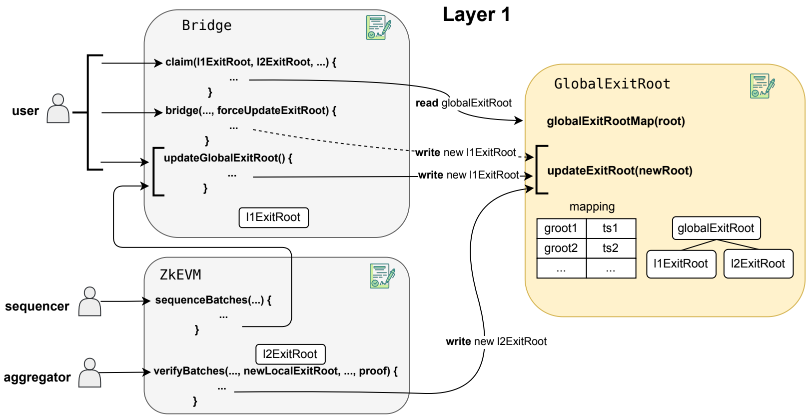
Sequencing


SEQUENCE OF n+1 BATCHES (A call to `sequenceBatches()` with a common L2 coinbase)
BATCH$_i$
timestamp$_i$
globalExitRoot$_i$
L2 TX's
BATCH$_{i+n}$
timestamp$_{i+n}$
globalExitRoot$_{i+n}$
L2 TX's
. . .
accInputHash$_{i-1}$
accInputHash$_i$
accInputHash$_{i+n-1}$
accInputHash$_{i+n}$
hash
hash
hash
hash
batchHashData$_i$
timestamp$_i$
globalExitRoot$_i$
L2 coinbase
oldAccInputHash
newAccInputHash
batchHashData$_{i+n}$
timestamp$_{i+n}$
globalExitRoot$_{i+n}$
L2 coinbase
oldAccInputHash
newAccInputHash
L1 ZkEVM contract checks:
$k > \ell$ timestamp$_k$ $\geq$ timestamp$_\ell$
timestamp$_k$ $\leq$ L1Block.timestamp
$\exists$ globlalExitRoot$_k$

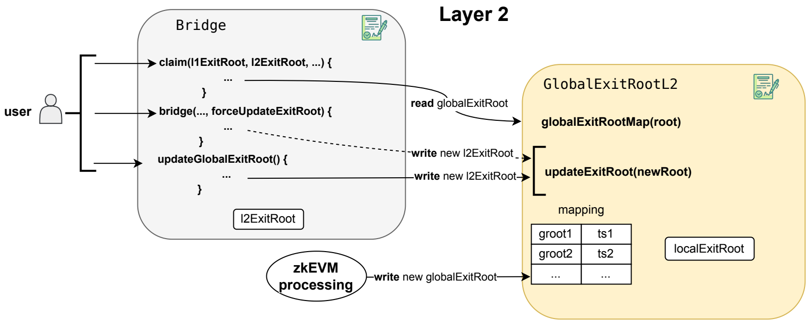
Proving and Sequencing



globalExitRoot in L1

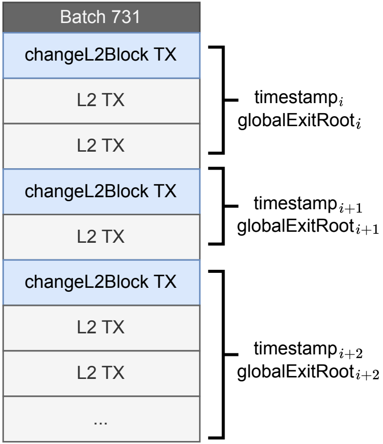


globalExitRoot in L2



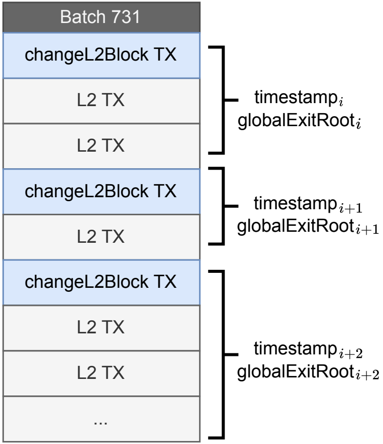
Outline

L2 Blocks and the changeL2Block Transaction



- In the fork-dragonfruit, the blocks within a batch share the same timestamp and `globalExitRoot`.
- In fork-etrog, each block has:
  - Its own `timestamp`.
  - Its its own `globalExitRoot`.
- These parameters are provided by the sequencer in the ”special” transaction `changeL2Block`.
- Each batch must start with a `changeL2Block` transaction.

Moving Checks from L1 to zkEVM Processing



- In fork-dragonfruit, the checks over the Batch's **`timestamp` bounds** and the **`globalExitRoot` existence** were performed by the L1 ZkEVM smart contract.
- In fork-etrog, notice that since we have a different timestamp (and possibly `globalExitRoot`) per L2 block, we have much more checks to perform.
- To decrease L1 costs, we move the checks of the **globalExitRoot existence** and the **timestamp bounds** to the zkEVM processing.
- That is, these checks are included in the proof and removed from the ZkEVM L1 smart contract.

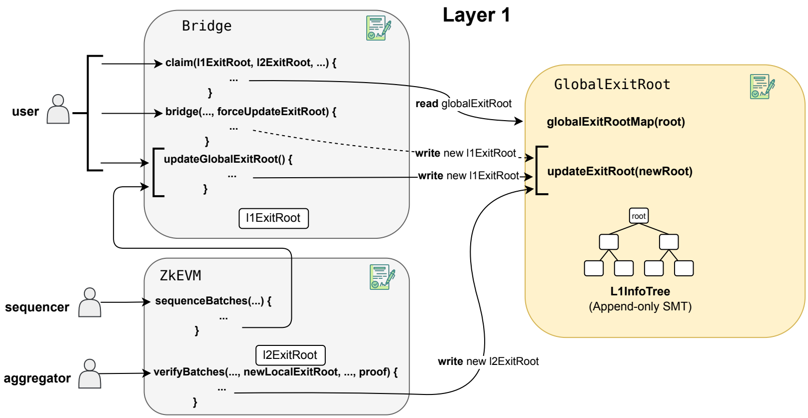
Checking the `globalExitRoot` Existence at the zkEVM Processing

`globalExitRoot` existence

To check the existence of a globalExitRoot, the zkEVM proving system needs to **have access to all the globalExitRoots recorded at L1**.

- However:
  - The `globalExitRoot`s are stored in a mapping in L1, at the `GlobalExitRootManager` smart contract, however, we cannot pass a mapping to the prover.
  - We could think in passing the list of `globalExitRoot`s but this is inefficient since this list is a potentially big and always growing data structure.
- The best way is to build a Merkle tree with all the `globalExitRoot`s.
- This tree is called the **L1InfoTree**.

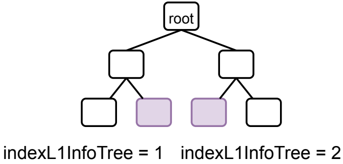
globalExitRoot in L1: L1InfoTree



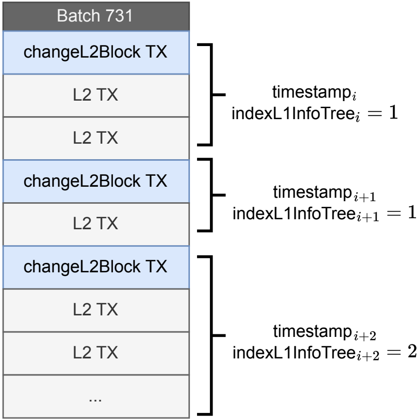
The **L1InfoTree** is an append-only SMT with same implementation as exit trees that is updated with new **globalExitRoots** by L1 **GlobalExitRoot** contract (replaces mapping of fork-dragonfruit).

Proving Batches using the L1InfoTree i

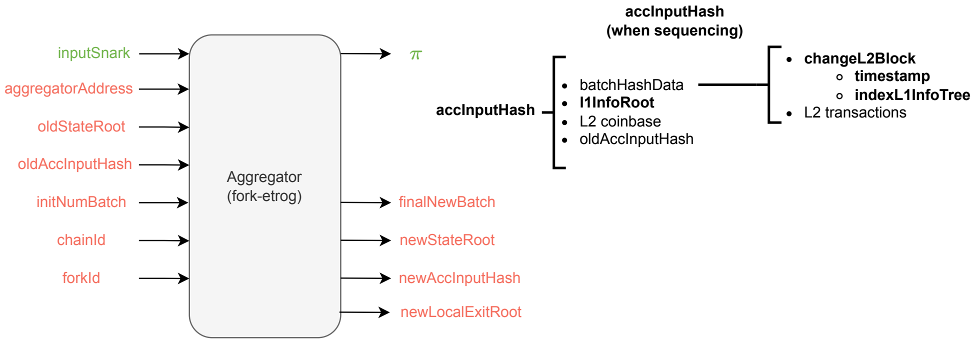
- The prover needs to have access to:
  1. The root of the L1InfoTree.
  2. The index of the `globalExitTree` being used for processing the L2 block.
- The index of the `globalExitTree` being used is called `indexL1InfoTree` and it is provided in the `changeL2Block` transaction.



L1InfoTree
(Append-only SMT)



Proving Batches using the L1InfoTree ii

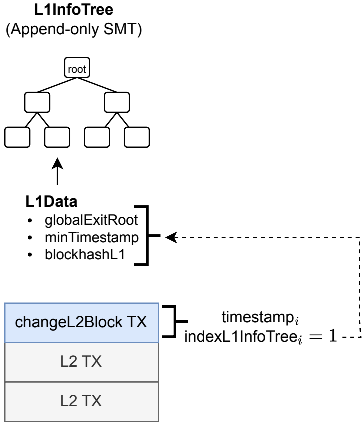


Note. Aggregator needs to know the SMT proofs given (`indexL1InfoTree`, `l1Inforoot`) to compute the proof.

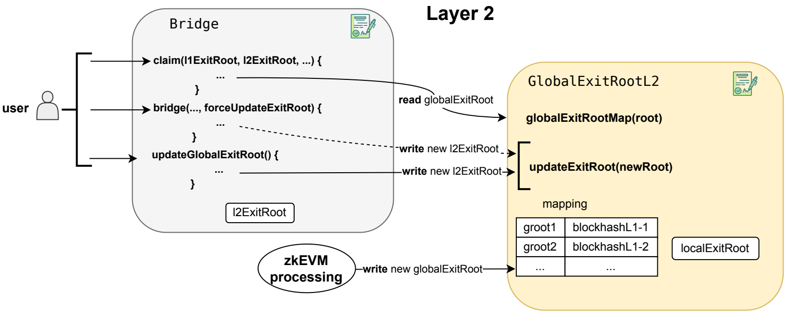
L1Data and the blockhashL1

In fact, each leaf of the L1InfoTree stores the following information:

- globalExitRoot.
- minTimestamp:
    - It is the time at which the globalExitRoot was recorded.
    - It is used by the timestamp checks as the minimum timestamp possible for a block (explained next).
- blockhashL1:
    - Blockhash of the L1 block that precedes the block in which it is placed the transaction that inserts the globalExitRoot in the L1InfoTree.
    - Recall that the header of an Ethereum block includes the (L1) state root, so making available the blockhashL1 provides the L1 state to L2 contracts.

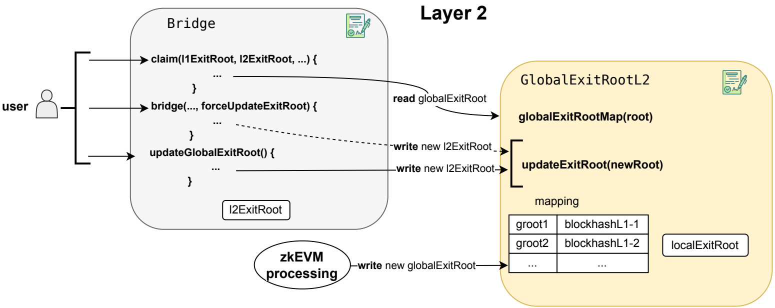


globalExitRoot in L2 i



- The zkEVM processing inserts the new **globalExitRoots** in the mapping of the **GlobalExitRootL2** contract, however, the mapping **is not updated** if:
    1. The **globalExitRoot** is already inserted or,
    2. The **indexL1InfoTree** is 0 (the first element of the tree does not contain data but its index has the special purpose of not upgrading and saves gas and data availability).

`globalExitRoot` in L2 ii



- Notice that unlike in fork-dragonfruit in which the mapping stored timestamps, in fork-etrog we store the **`blockhashL1`** associated with the **`globalExitRoot`**.
- The **`blockhashL1`** is also stored as part of the **`blockhashL2`**, which provides a summary of the execution of the L2 block including the current L2 state.
- The **`blockhashL1`** can be used by L2 transactions, during their processing, to access L1 data.

The **changeL2Block** Transaction i

Field Name	Size
type	1 byte
deltaTimestamp	4 bytes
indexL1InfoTree	4 bytes

The `type` field:

- It is used to distinguish the `changeL2Block` transaction from regular L2 transactions.
- The value used is `0x0C`, while regular L2 transactions are rlp-encoded and their first byte is always bigger than `0xC0`.
- We also leave room for Ethereum typed transactions which use low values in their type field.

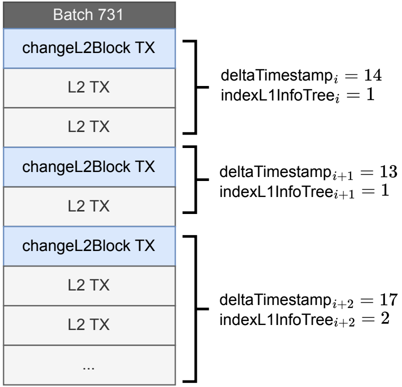
The `indexL1InfoTree` field:

- This is the index of the `globalExitRoot` being used by the block.
- The `L1InfoTree` has 32 levels, that is, keys of 32 bits (4 bytes).
- Recall that 0 has the special meaning of not updating in L2.

The `changeL2Block` Transaction ii

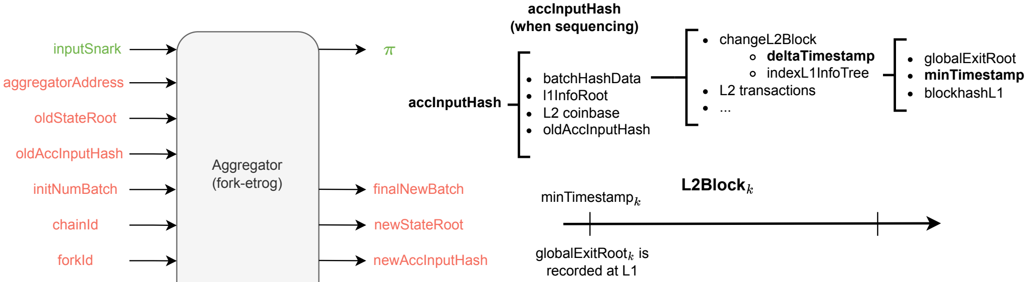
The `deltaTimestamp` field:

- Instead of using absolute timestamps, we use incremental timestamps.
- The `deltaTimestamp` shows the amount of seconds that need to be **added to the timestamp of the previous L2 block** to obtain the timestamp of the current block.
- The `timestamp` of the previous L2 block is available to the zkEVM in the system contract `0x5ca1ab1e` as part of the `blockhashL2`.



Note. We use incremental timestamps to reduce the size of this field (data availability): using a regular Unix time timestamp we would need to use 64 bits, while increments are always much smaller and we use 32-bits.

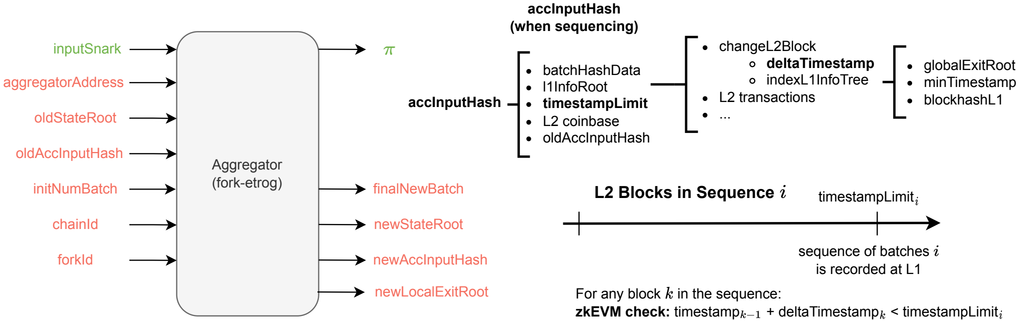
Checking the Lower timestamp Bound



For any block k in the sequence:

zkEVM check: timestamp$_{k-1}$ + deltaTimestamp$_k$ > minTimestamp$_k$

Checking the Upper `timestamp` Bound



For any block k in the sequence:

zkEVM check: timestamp$_{k-1}$ + deltaTimestamp$_k$ < timestampLimit$_i$