polygon zkEVM

Knowledge Layer

Architecture

Layer Communication up to fork-6 (etrog)

v.1.0

May 23, 2024

1 Introduction

In this section we will delve in the implementation of the layer communication in fork-6. Recall that in this version we have different blocks inside a batch and each one with different transactions, unlike in the fork-5 version, where a block and a transaction were the same.

An important alteration lies in the fact that each block now possesses its individual timestamp and `globalExitRoot`, as illustrated in Figure 1. These parameters are provided by a special transaction known as `changeL2Block`, tasked with marking the transition between blocks. It's worth noting that in etrog, every block and batch initiation commences with the execution of a `changeL2Block` transaction.

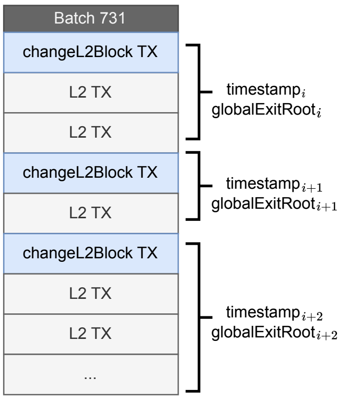

Figure 1: In this batch, multiple blocks are present, and the transition between them is delineated by the `changeL2Block` transaction. Unlike the previous version (dragonfruit), each block now accommodates one or more transactions, with each block being assigned its distinct `globalExitRoot` and timestamp.

2 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. However, in fork-etrog, due to the presence of distinct timestamps (and possibly `globalExitRoots`) per L2 block, a greater number of checks are required. To decrease L1 costs, we opt to transfer the verification of `globalExitRoot` existence and **timestamp bounds** to the zkEVM processing. Consequently, these checks are incorporated into the proof and removed from the zkEVM L1 smart contract.

Recall that in order to check the existence of a `globalExitRoot`, the zkEVM proving system would need to have access to all the `globalExitRoots` recorded in L1, which are stored in a mapping within the `GlobalExitRootManager`. However, we can not pass a mapping to the prover. A naive solution would be to pass a list of `globalExitRoots` to the prover, but this is highly inefficient since this list is a potentially big and always growing data structure. The best way it to build a Merkle tree with all the `globalExitRoots`. We will refer to this tree as the **L1InfoTree**.

The **L1InfoTree** is an append-only SMT with same implementation as exit trees and it is updated with new `globalExitRoots` by L1 `GlobalExitRoot` contract. So, it replaces the mapping of fork-dragonfruit, as we can see in Figure 2.

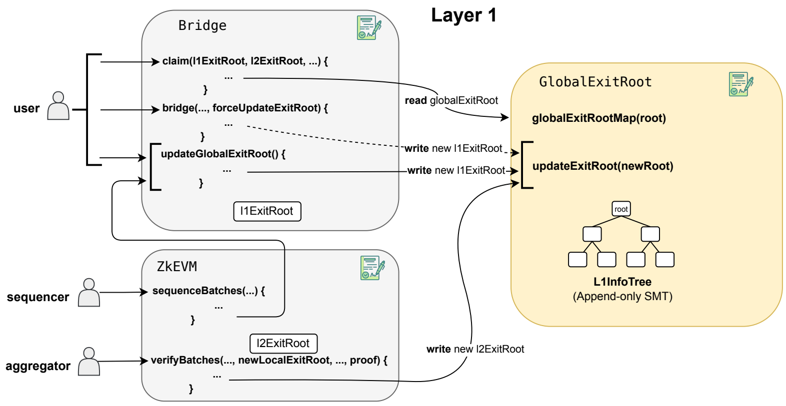

Figure 2: In fork-etrog the `globalExitRoots` are stored in a Merkle Tree called **L1InfoTree** instead of in a mapping.

3 Proving Batches using the L1InfoTree

To prove the existence of a specific leaf using an append-only tree-like structure to store the `globalExitRoots`, the prover must have access to both the root of the L1InfoTree and the index of the `globalExitRoot` being used for processing the L2 block (See Figure 3). The index of the `globalExitRoot` being used is called `indexL1InfoTree` (See Figure 4) and it is provided in the `changeL2Block` transaction.

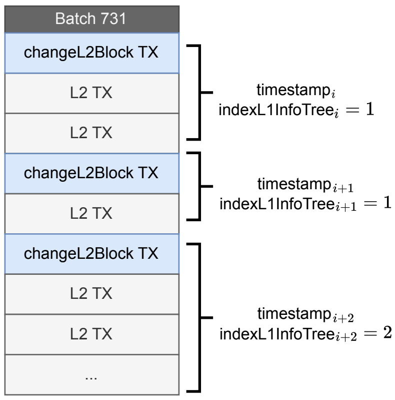

Figure 3: As we can observe, in the first block, we obtain its corresponding timestamp and its index, which is 1. The same applies to the second block; it has its own timestamp, and its `globalExitRoot` is in the same leaf as the previous block since it also has an index of 1. However, the `globalExitRoot` of the third block has an index of 2, so it is in another leaf of the `L1InfoTree`.

Recall that in fork-dragonfruit each batch had its own timestamp and `globalExitRoot` so this parameters were contained in `accInputHash` at batch level, and within `batchHashData` we only had the hash of the corresponding L2 transactions. In fork-etrog (See Figure 5), since the timestamp is defined at the transaction level, we need to include it in the `batchHashData`, along with the `indexL1InfoTree` and the previously included transactions. Observe that, since the tree is incremental within a batch, the root of the L1InfoTree can be sent at batch level.

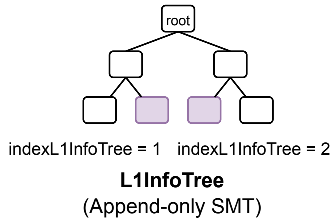

Figure 4: To provide a proof for an append-only Merkle tree, the prover needs to know the root and the index of the leaf that the verifier is trying to access.

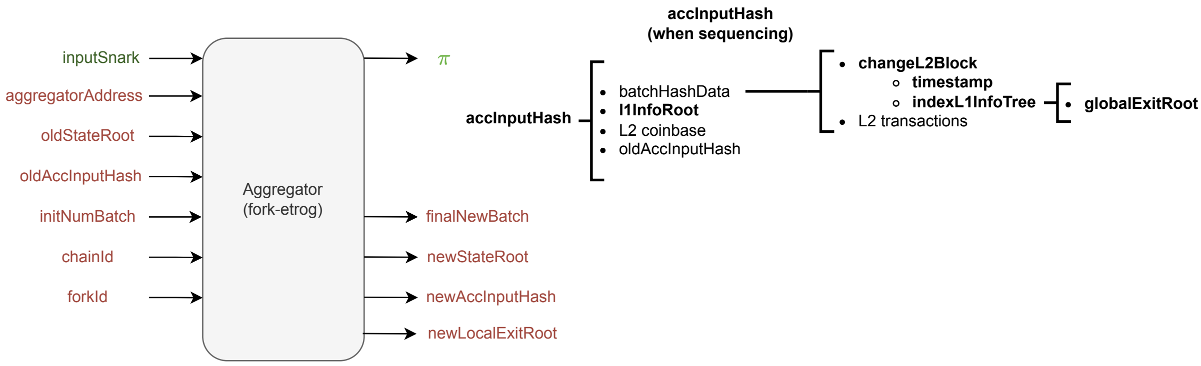

Figure 5: We can see how the parameters contained in `accInputHash` have changed with respect to the previous version.

4 Data of the L1InfoTree

In fact, the L1InfoTree, besides storing the `globalExitRoots`, also contains two other parameters, as shown in Figure 6:

The first parameter is the `minTimestamp`, which represents the time when the `globalExitRoot` was recorded in the tree. This parameter will be used in the timestamps checks as the minimum timestamp possible for a block. We will deep into this later on.

The other parameter is the `blockhashL1` which is the 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.

5 GlobalExitRoot in L2

The zkEVM processing inserts the new `globalExitRoots` in the mapping of the `GlobalExitRootL2` contract (see Figure 7), however, the mapping **is not updated** if the `globalExitRoot` is already inserted or, if the `indexL1InfoTree` is 0 (the first element of the tree does not contain data but its index has the special purpose of not upgrading the `globalExitRoot` in L2, which saves data availability and ZK processing).

Unlike fork-dragonfruit, where the mapping stored timestamps, in fork-etrog, we store the `blockhashL1` associated with the `globalExitRoot` (see Figure 7). 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.

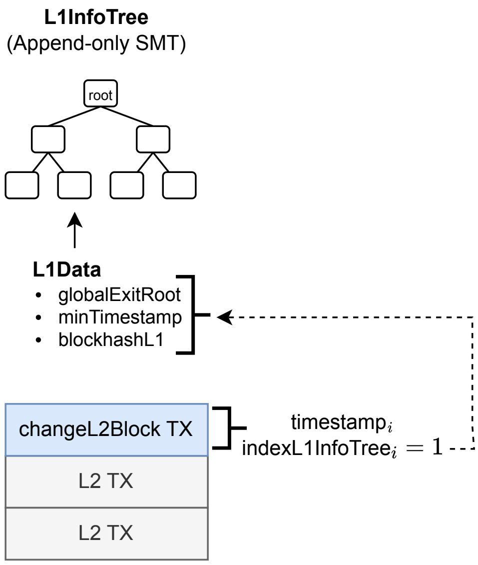

Figure 6: When a `changeL2BlockTx` is executed, the corresponding L1InfoTree leave is not only populated with `globalExitRoots`, but also with two important parameters: the `minTimestamp` and the `blockhashL1`

6 The `changeL2Block` Transaction

We've already introduced the `changeL2Block` transaction, a novel addition introduced in etrog. Its primary function is to mark the transition between blocks within a batch, with each batch and block commencing with this transaction. However, it also provides metadata from the block.

As we can observe in the following figure, the `changeL2Block` Transaction has 9 bytes, distributed in the following way:

The `type` field 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 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 `deltaTimestamp` field 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 (see Figure 9). So, instead of using absolute timestamps, we use **incremental timestamps** in order to reduce the size of this field for cheapen data availability: using a regular Unix timestamp, we would need to use 64 bits, while increments are always much smaller, so we use 32-bits. The `timestamp` of the previous L2 block is available to the zkEVM in the system contract `0x5ca1ab1e` as part of the `blockhashL2`.

7 Checks by the zkEVM Processing

7.1 Lower timestamp Bound

This check occurs when the zkEVM initiates the processing of a block. The objective is to verify that the timestamp of the block is greater than the `minTimestamp`, which

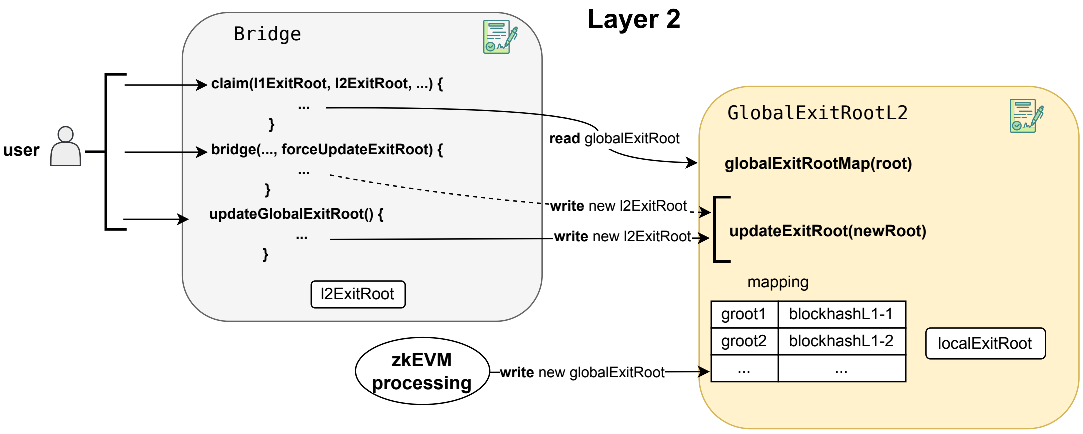

Figure 7: The new mapping (etrog) for the global exit roots in the `GlobalExitRootL2` smart contract relates each `globalExitRoot` with the corresponding block hashes of L1 instead of the timestamp.

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

Figure 8: The data structure of the `changeL2Block` transaction.

corresponds to the timestamp of the `globalExitRoot` utilized by this block. As previously mentioned, the `minTimestamp` is contained within the data of the L1InfoTree, and the prover retrieves it via the `indexL1InfoTree` (see Figure 10).

7.2 Upper timestamp Bound

For this check, a new parameter, `timestampLimit`, is introduced in the `accInputHash`. This parameter represents the timestamp of the transaction calling the function `sequenceBatches`, which sequences multiple batches. In the Figure 11, we can observe the complete set of data received by the zkEVM processing in the etrog fork.

The objective is to verify that the timestamp of each block within these batches is earlier than the `timestampLimit`. In other words, we ensure that the blocks were created before being sequenced.

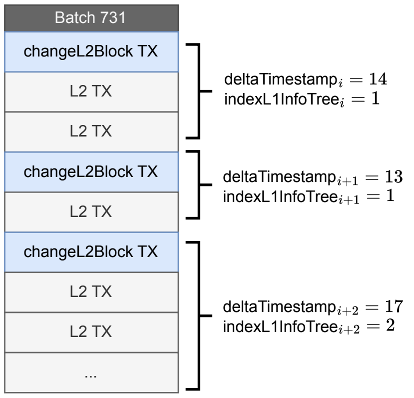

Figure 9: The delta timestamp of the second block is 13, which means that the timestamp of this block is 13 seconds later than the timestamp of the previous block.

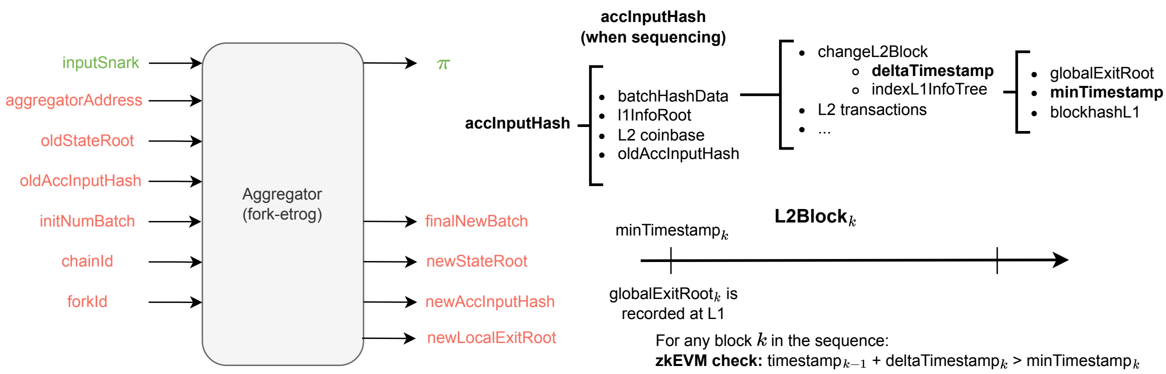

Figure 10: We can see in this figure the parameters that have had to be added in `accinputhash` to be able to carry out the lower timestamp bound check.

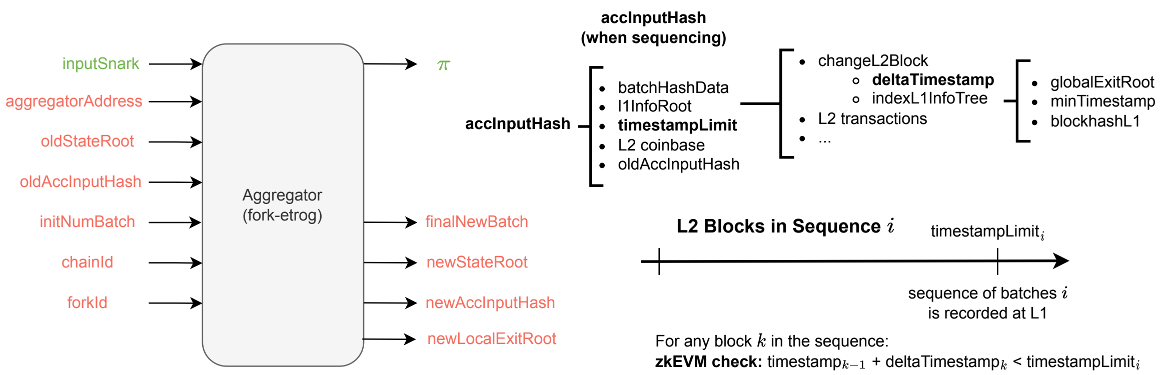

Figure 11: This is the final configuration of the `accInputHash` parameter in fork-6.