Knowledge Layer

Architecture

Layer Communication up to fork-6 (etrog)

v.1.0

May 24, 2024

1 Introduction

In this section we will delve in the changes that has been introduced in the layer communication domain by the fork-etrog. Recall that previously to fork-etrog, a single transaction was the same as a L2 block. With the introduction of this new fork, L2 blocks can consists on one or more transactions. The boundaries of a L2 block are recorded using a new transaction called called `changeL2Block` which is in charge of marking a transition between L2 blocks.

One of the main issues solved with the introduction of this new fork is the possibility to define a timestamp at a block level, as illustrated in Figure 1. Previously, the timestamp was defined at batch level, which was set by the sequencer when the batch started to being filled with transactions. Now, since each block should have its own timestamp, the associated change-of-block transaction has to incorporate information about the starting timestamp of the block. In a similar fashion, each proved block within a batch should have associated its own global exit root, which is decided by the sequencer as before. Observe that, unlike the timestamp, the global exit root can be equal across multiple consecutive blocks. This information is also sent in the `changeL2Block` transaction, altogether with the corresponding timestamp. This new paradigm introduces changes into the layer communication domain. For instance, we can no longer use a global exit root for a block with an earlier timestamp than the one where the global exit root is updated.

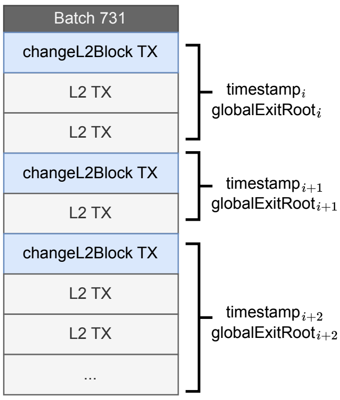

Figure 1: In this batch, multiple blocks are present, and the transition between them is delineated by the `changeL2Block` transaction. Unlike the previous fork, each block now accommodates one or more transactions, with each block being assigned its distinct `globalExitRoot` and timestamp. It's worth noting that in etrog, every block and batch initiation commences with the execution of a `changeL2Block` transaction.

2 The `L1InfoTree`

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 global exit roots) 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 global exit roots 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 global exit roots to the prover, but this is highly inefficient since this list is a potentially big and always growing data structure. A succinct way to do it is to build a Merkle tree with all the global exit roots. 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 global exit roots by the L1 `GlobalExitRoot` contract. Henceforth, it replaces the mapping of fork-dragonfruit, as we can see in Figure 2.

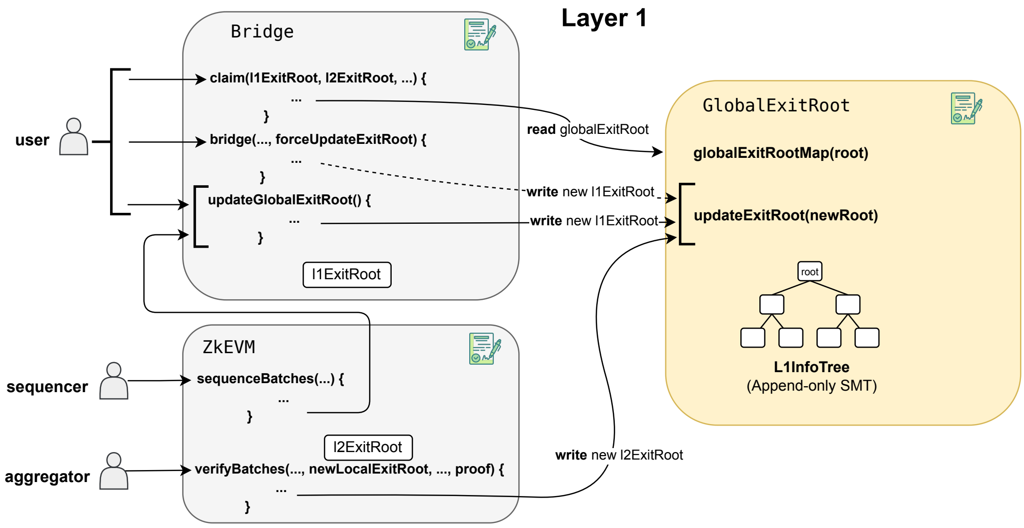

Figure 2: In fork-etrog the `globalExitRoots` are stored in a Merkle Tree called `L1InfoTree` instead of in a mapping.

To generate a proof for the existence of a specific leaf using an append-only tree-like structure that stores all the global exit roots, the prover must have access to both the root of the `L1InfoTree` and the index of the `globalExitRoot` used for processing the L2 blockThis index, called `indexL1InfoTree`, serves to locate the specific leaf within the tree. Observe that providing the prover with both the `indexL1InfoTree` and the corresponding `globalExitRoot` is redundant. Therefore, instead of providing both in the `changeL2Block` transaction, we will only provide the `indexL1InfoTree`, which completely determines the used global exit root.

With these modifications, we also need to adjust the information provided to the aggregator to ensure correct aggregation of proofs. 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 4), 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. Moreover, we should include the last root of the `L1InfoTree` within the batch, which will be called `L1InfoRoot`. Observe that this is enough since the `L1InfoTree` is incremental, so all the inclusion proofs can be generated using the lastly updated root.

In fact, the leaves of the `L1InfoTree`, besides storing the global exit roots, also contains two other parameters, the `minTimestamp` and the `blockhashL1`, as shown in Figure 5. The summary of the three parameters is called `L1Data`.

- 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.

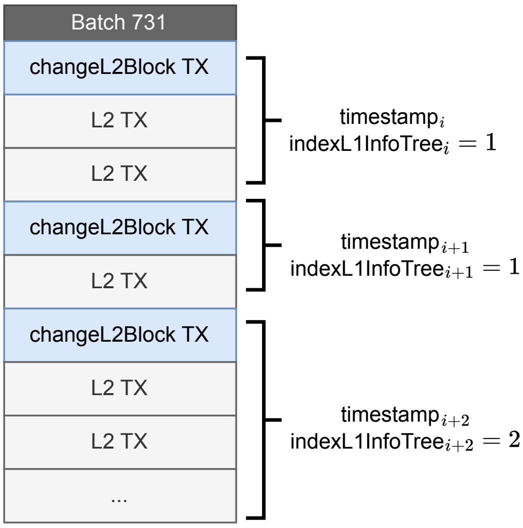

Figure 3: As we can observe, in the first block, we obtain its corresponding timestamp and its index, which is 1. The same applies to the second block; it has its own timestamp, and its `globalExitRoot` is in the same leaf as the previous block since it also has an index of 1. However, the `globalExitRoot`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

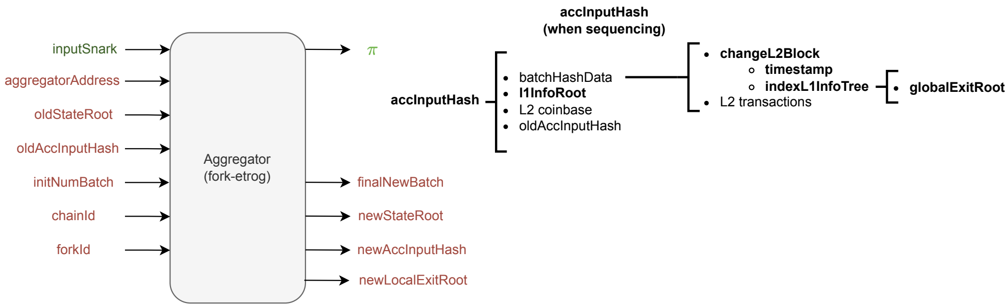

Figure 4: We can see how the parameters contained in `accInputHash` have changed with respect to the previous version.

- 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.

3 L2 Global Exit Root Management

Up to this point, we have discussed the management of the global exit root in L1, which is handled by the `GlobalExitRoot` smart contract. However, it is important to note that the global exit root in L2 is managed by a different smart contract called `GlobalExitRootL2`. This contract maintains a similar mapping structure to the one used in fork-dragonfruit to store the global exit roots. During the zkEVM processing, the new `globalExitRoot` used by a certain block is inserted in the mapping, as shown in Figure 6. However, the mapping **is not updated** if the `globalExitRoot` is already inserted or, if the `indexL1InfoTree` is 0, meaning that the first element of the tree does not contain data but its index has the special

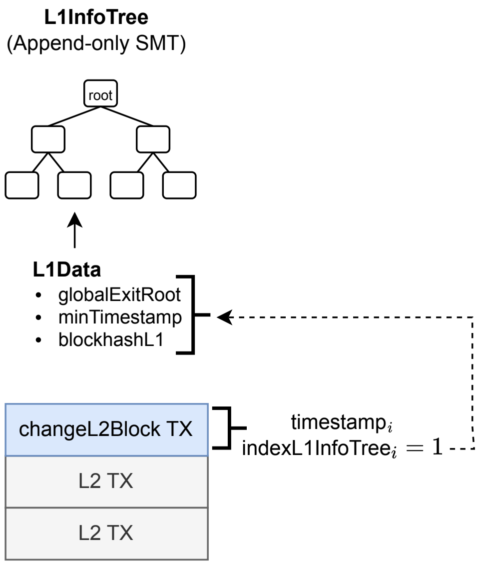

Figure 5: When a `changeL2BlockTx` is executed, the corresponding L1InfoTree leave is not only populated with `globalExitRoots`, but also with two important parameters: the `minTimestamp` and the `blockhashL1`

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`. 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.

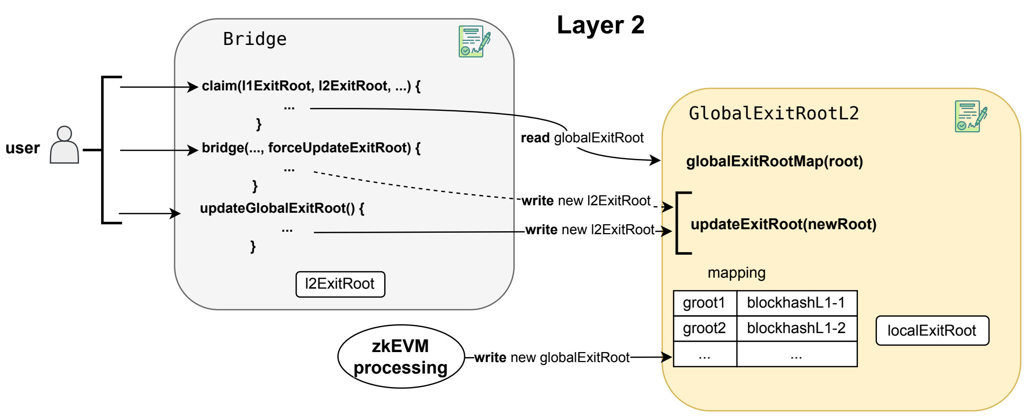

Figure 6: The new mapping (etrog) for the global exit roots in the `GlobalExitRootL2` smart contract relates each `globalExitRoot` with the corresponding block hashes of L1 instead of the timestamp.

4 The `changeL2Block` Transaction

We've already introduced the `changeL2Block` transaction, a novel addition introduced in etrog. Its main function is to mark the transition between blocks within a batch. Recall that, as commented before, this transaction should provide information such as the timestamp and the `indexL1InfoTree`. The data structure of the `changeL2Block` transaction is illustrated in Figure 7.

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

Figure 7: The data structure of the `changeL2Block` transaction.

To differentiate this special transaction from regular L2 transactions, we will introduce a `type` field as the first byte of the transaction. Regular L2 transactions are RLP-encoded, and their first byte is always different from `0x0B`. Therefore, we will use `0x0B` as the unique identifier for this special transaction type.

Recall that the `indexL1InfoTree` field is the index of the `globalExitRoot` being used by the block. The `L1InfoTree` has 32 levels, that is, its keys consists on 32 bits (or equivalently, 4 bytes). Note that 0 has the special meaning of not updating in L2.

While we previously stated that the timestamp must be included in the `changeL2Block` transaction, there is an alternative strategy to reduce data costs. Instead of using absolute timestamps, we employ **incremental timestamps** to minimize the size of this field, thus lowering data availability costs. A standard Unix timestamp requires 64 bits, whereas increments are much smaller, allowing us to use just 32 bits. This incremental timestamp, called `deltaTimestamp`, represents the number of seconds to be added to the timestamp of the previous L2 block to determine the current block's timestamp (see Figure 8). The timestamp of the previous L2 block is accessible to the zkEVM via the system contract `0x5ca1ab1e` as part of the `blockhashL2`. As we will later discuss, this, combined with the `minTimestamp`, will be sufficient for the prover to verify timestamp bounds.

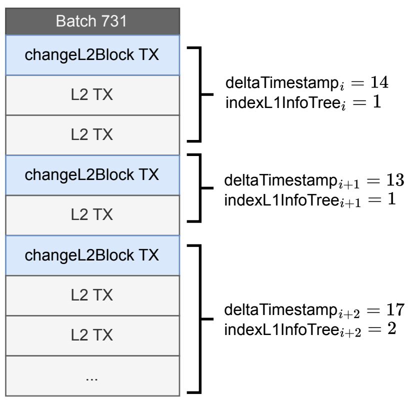

Figure 8: The delta timestamp of the second block is 13, which means that the timestamp of this block is 13 seconds later than the timestamp of the previous block.

5 Timestamp Checks

In the zkEVM, ensuring accurate timestamp management is crucial for maintaining the integrity of block processing. Two critical checks are implemented to validate timestamps: the lower timestamp bound check and the upper timestamp bound check. These checks ensure that blocks are processed within the correct time frames, preventing inconsistencies and potential security issues.

5.1 Lower Timestamp Bound

This check occurs when the zkEVM initiates the processing of a block. The aim of this check is to verify that the block's timestamp is greater than the `minTimestamp`, which corresponds to the timestamp of the `globalExitRoot` utilized by this block.

$$\texttt{timestamp}_{k-1} + \texttt{deltaTimestamp}_k > \texttt{minTimestamp}_k$$

As previously mentioned, the `minTimestamp` is contained within the data of the `L1InfoTree`, and the prover retrieves it via the `indexL1InfoTree`. Observe that we should include the `deltaTimestamp` to the accumulated input hash as shown in Figure 9.

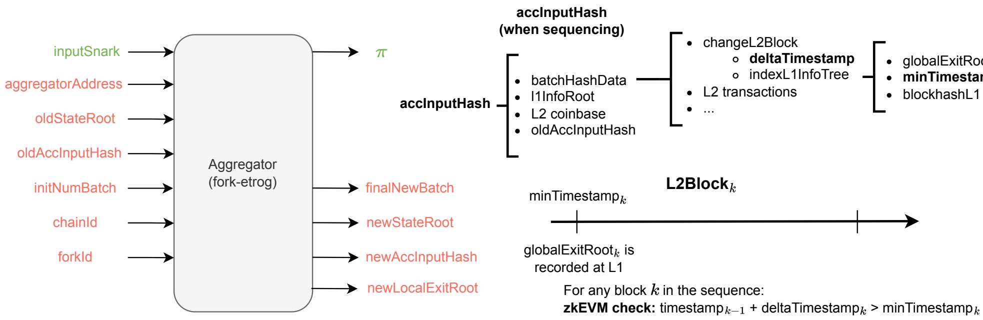

Figure 9: The accumulated input hash should be modified in order to include the `deltaTimestamp`.

5.2 Upper Timestamp Bound

The aim of this check 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. For this check, a new parameter, `timestampLimit`, is introduced in the `accInputHash`, as shown in Figure 10. This parameter represents the timestamp of the transaction calling the function `sequenceBatches`, which sequences multiple batches.

$$\texttt{timestamp}_{k-1} + \texttt{deltaTimestamp}_k < \texttt{timestampLimit}_k$$

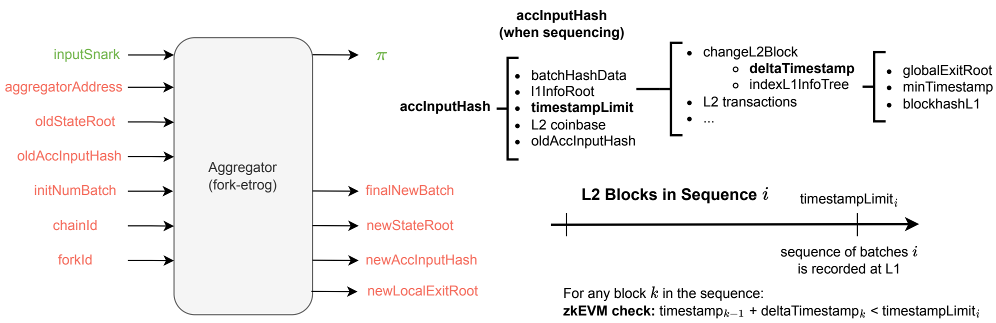

Figure 10: This is the final configuration of the `accInputHash` parameter in fork-6.