polygon zkEVM

Knowledge Layer

Architecture

Processing L2 Blocks

v.1.0

May 22, 2024

1 Introduction

In this section, we'll discuss the differences between fork-5 (dragonfruit) and fork-6 (etrog), mainly related to the definition of a block.

In fork-dragonfruit, a block corresponds to a transaction, resulting in as many blocks per batch as there are transactions. Recall that the timestamp is part of the batch data and, henceforth, is shared for all the blocks within the batch. While dragonfruit's approach minimizes delay, it presents several challenges: it leads to significant database data due to the large number of L2 blocks created and lacks a mechanism to assign different timestamps to individual blocks within a batch. This deficiency disrupts many Dapps reliant on timestamps to define the timing of actions performed by smart contracts.

The fork-etrog approach addresses these issues by enabling multiple transactions per block, which can be achieved by using a small timeout of seconds or miliseconds when creating the block for waiting for transactions. Moreover, the sequencer can change the timestamp of the different blocks within the batch by using a special transaction or marker called `changeL2Block`. In Figure 1 we can observe how it works.

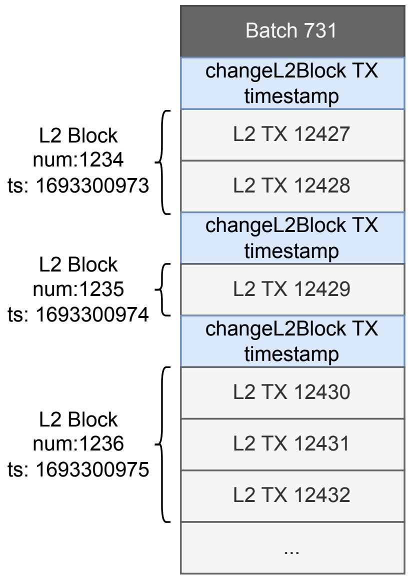
Batch 731
changeL2Block TX timestamp
L2 Block num:1234 ts: 1693300973
L2 TX 12427
L2 TX 12428
changeL2Block TX timestamp
L2 Block num:1235 ts: 1693300974
L2 TX 12429
changeL2Block TX timestamp
L2 Block num:1236 ts: 1693300975
L2 TX 12430
L2 TX 12431
L2 TX 12432
...

Figure 1: Fork-6 new approach. A new special kind of transaction called `changeL2Block` is introduced in the batch to mark whenever there is a block change. This special transaction is in charge of changing the `timestamp` and the L2 `blocknumber`.

In this document, we'll delve into the processing of a block in both fork-dragonfruit and fork-etrog.

2 The `0x5ca1ab1e` Smart Contract

We will start this document from the point of view of the proving system. Recall that the aggregator received several inputs, such as the `oldStateRoot`, the `initNumBatch` and the `oldAccInputHash`. Also recall that the accumulated input hash `accInputHash` is an recursive cryptographic representative (that is, a hash) of several batch data, such as a hash of all the L2 transactions within the batch or the batch's sequencing timestamp. Given its recursive nature, the accumulated input hash also incorporates the previous accumulated input hash as part of its hashed data.

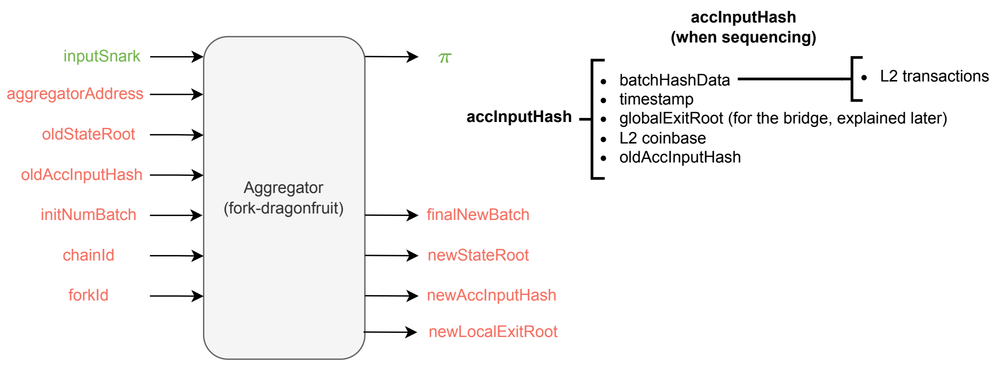

Figure 2: Aggregator schema in fork-dragonfruit.

As illustrated in Figure 2, one might wonder how the prover's processing system is able to access and secure the block number (which, in the context of fork-dragonfruit, is equivalent to the transaction number). The explanation is that this information is included within the L2 state. To be more specific, this data is held in the storage slot 0 of a L2 system smart contract, which is deployed at the address `0x5ca1ab1e`. After the processing a transaction, the ROM writes the current block number into this specific storage location, as shown in Figure 3. Henceforth, during each batch processing, the state records all block numbers it contains.

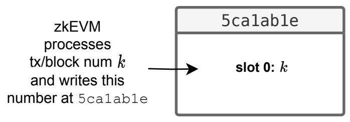
zkEVM processes tx/block num k and writes this number at 5ca1ab1e
5ca1ab1e
slot 0: k

Figure 3: L2 system smart contract `0x5ca1ab1e` storing at slot 0 the number of the last block processed by the zkEVM.

Throughout this document, the smart contract deployed at the address `0x5ca1ab1e` will be frequently referenced. For simplicity and convenience, we will abuse of notation and often refer to it as the `0x5ca1ab1e` smart contract.

3 The `BLOCKHASH` Opcode

In the L1 EVM, the `BLOCKHASH Opcode` provides the `keccak-256` of the **Ethereum L1 block header** (See Figure 4) includes information like: coinbase address, block number, timestamp, root of the state trie, root of transactions trie, root of receipt trie, difficulty, gas limit, gas used, logs, etc. Here you can find a list of all of them. Here you can find the Geth library to compute the block hash in the Ethereum way.

Our RPC provides methods for obtaining the L2 `BLOCKHASH` in the Ethereum-style. Recall that you can find a list of specific zkEVM endpoints here. One such method is `zkevmget_FullBlockByHash`, which retrieves a block with additional information given its Ethereum-like block hash.

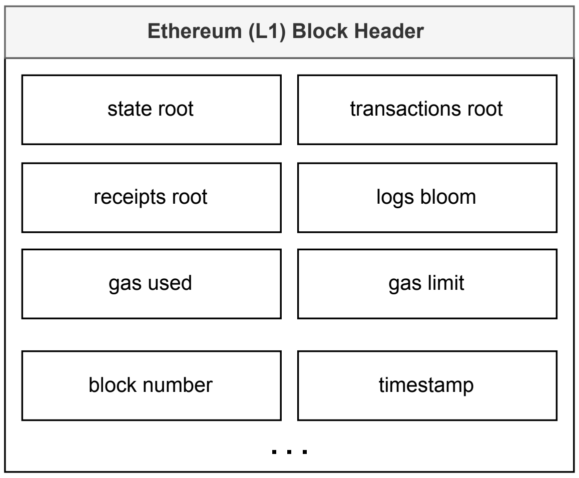

Figure 4: An example of the Ethereum L1 block header with some parameters.

4 Fork-dragonfruit (fork-5)

4.1 L2 `BLOCKHASH`

Following Ethereum's philosophy, we should keep track every change of state between blocks, which is equivalent to tracking transactions up to fork-5. In order to secure this, we should include all of them into the state. We will do it storing the state root after each transaction processing in the slot 1 of the `0x5ca1ab1e` smart contract, as shown in Figure 5. Observe that this process is repeated numerous times during batch processing, enabling the state to accurately monitor the entirety of batch processing at transaction level.

In fork-5, when the `BLOCKHASH` Opcode is executed by smart contracts, it provides **only the state root**. We define this particular output of the `BLOCKHASH` opcode as **native block hash**. We provide the state root accessing the `0x5ca1ab1e` smart contract.

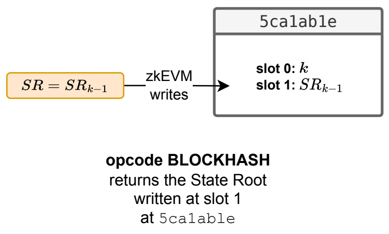

Figure 5: L2 system smart contract `0x5ca1ab1e` storing at slot 1 the last state root after processing a block.

Up to `fork-dragonfruit` included, the zkEVM processing does not store extra parameters about the block execution inside `0x5ca1ab1e`.

4.2 Processing L2 Blocks

The next topic to be addressed is how to write the state root in the `0x5ca1ab1e` system smart contract. The explanation will follow Figure 6 in order to ease the understanding.

Let us suppose that we have completed processing the last block of a batch which we will denote as block $k-1$. We will refer to the state root at this point by SR_{k-1}. Then,

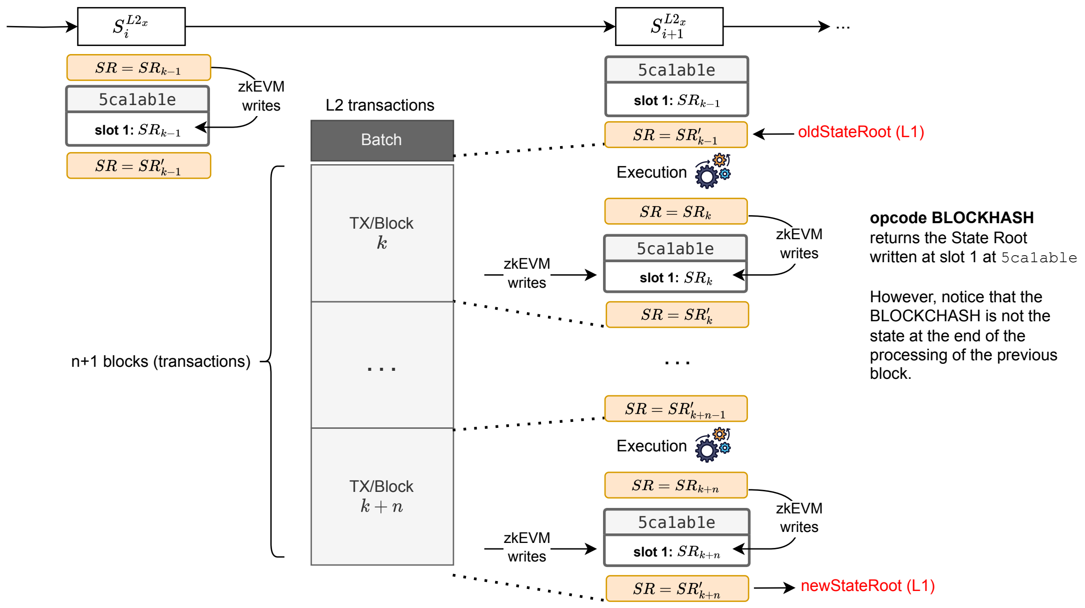

Figure 6: This is a schema of how the new state root is updated and written in the `0x5ca1ab1e` system smart contract when processing L2 blocks. The process is explained below.

the zkEVM updates the slot 1 of the `0x5ca1ab1e` smart contract with the current state root. Since we are updating a storage slot of a L2 smart contract, we are changing the L2 state, leading to a new state, which we will denote SR'_{k-1}.

At this point, we start processing the new batch. It is important to notice that the state root SR_{k-1} before starting to process the new block and the state root stored in the `0x5ca1ab1e` contract SR'_{k-1} do not match. When the new block k is executed, the state root becomes SR_k, but again, upon writing it to the contract, the system becomes SR'_k. This pattern continues with subsequent blocks.

Therefore, relying on slot 1 as the root state of the system at the end of the execution of the previous block can be misleading because the actual corresponding state root is the one after updating the contract. Consequently, when `BLOCKHASH` is called in fork-5, the obtained value isn't precisely the state root just before starting the execution of the new block. This mismatch issue will be addressed in fork-etrog.

5 L2 Native `BLOCKHASH` vs. L2 RPC Ethereum-like `BLOCKHASH`

In Ethereum, the block header is secure beacuse it is computed and validated by all the nodes within the network. However, in the zkEVM, it is the prover who proves that the parameters related with the block execution are correct and these parameters are inserted in the state. In the Ethereum approach, the block parameters that provide information about the execution of the transaction inside the block are hashed to obtain the block hash and the resulting state root is one of these parameters. In the Polygon zkEVM, native block hash has to be the L2 state root because we want to prove that the blockhash computation and its parameters are correct. The zkEVM prover proves the L2 state root changes are correctly performed, so, if we want to provide a verifiable proof of the execution parameters of a block (gasUsed, transaction logs, etc.), we have to compute these parameters in the zkEVM processing and include them in the L2 state. The incorporation of block execution parameters into the L2 state is facilitated through the `0x5ca1ab1e` contract. Henceforth, the L2 state root is a hash that contains all the parameters that provide the information about the block execution. In Figure 7 this difference is graphically shown.

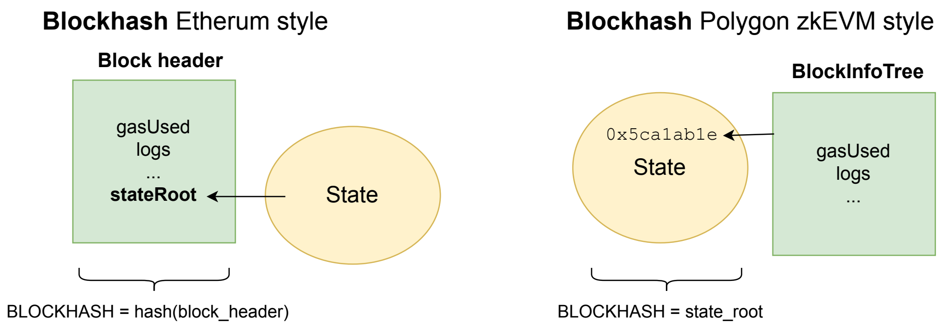

Figure 7: Blockhash in Ethereum and in zkEVM.

The native blockhash is computed differently than Ethereum-like blockhash. For example:

- As part of the block execution, we include zkEVM-specific parameters like the `effectivePercentage` for each transaction.
- We use `Poseidon` instead of `keccak-256`.
- Transactions data is hashed with a linear `Poseidon`.

In the RPC, the method `zkevm_getNativeBlockHashesInRange` returns the list of "native block hashes" i.e. a the list of L2 state roots.

6 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 `0x5ca1ab1e` stores:

- **Slot 0**: The last block number processed, as in fork-dragonfruit.
- **Slots `hash(1|blockNum)`**: These slots are encoded as a Solidity mapping to store all the state roots indexed per block number.
- **Slot 2**: The timestamp of the last block processed, because now we have a timestamp for each block.
- **Slot 3**: The root of a read-write Merkle tree called `BlockInfoTree` which contains information about the execution of the last processed block.

Figure 8 shows a schematic diagram of the `0x5ca1ab1e` contract's storage in fork-etrog.

6.1 `BlockInfoTree`

In this section, we will describe the `BlockInfoTree`, including its contents and its keys and values. As said before, the `BlockInfoTree` is a read-write Merkle tree containing information about the execution of the last processed block. Observe that this tree is unique for each block.

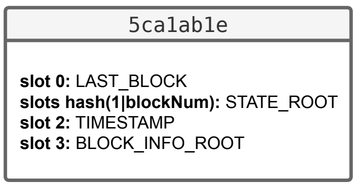

Figure 8: Slots of `0x5ca1ab1e` contract in fork-etrog.

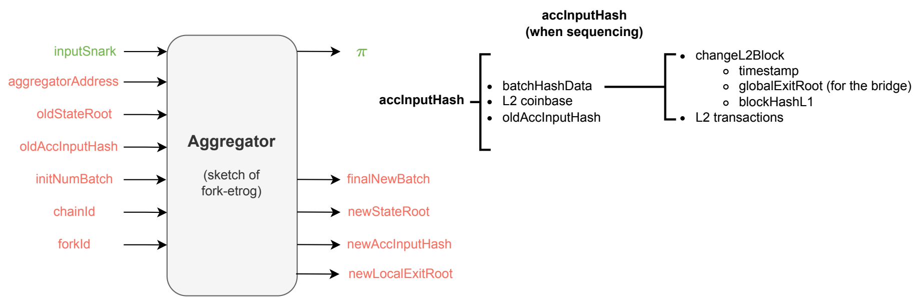

Figure 9: Aggregator schema in fork-etrog.

Contents Within the `BlockInfoTree` we store the **Transaction Data**, which refers to the specific data associated with a transaction. We will store a cryptographic representative of each L2 transaction data as follows:

$$\texttt{transactionHashL2} = \texttt{LinearPoseidon}(\texttt{txData})$$

being `txData` the following array of data (`nonce,gasPrice, gasLimit, to, value, data, from`). Recall that, actually, the `from` field is computed from the transaction's signature using ecRecover.

As we will see in subsequent documents, each finally executed transaction includes a byte parameter called `effectivePercentage`, which is zkEVM specific and is also included in the `BlockInfoTree`.

The `BlockInfoTree` also stores the following data that comes from L2 transactions' execution:

- `status`
- `receiptData`
- `cumulativeGasUsed`
- `linearPoseidon(log_0_data)`
- `linearPoseidon(log_1_data)`
- ...
- `linearPoseidon(log_N-1_data)`

In the tree there is also stored information that is common to all the block, consisting in the following dataw:

- Data stored/updated from L2 state, obtained from the `0x5ca1ab1e` smart contract:
  - `Previous blockhashL2` (that is, the previous L2 state root).
  - `blockNumber`.
- Data from each L2 block obtained from L1 as input for the proof:

- `timestamp` from the `changeL2block` transaction.
- `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 and Values We're tasked with storing all previous block-related and transaction-related data in the `BlockInfoTree`. This tree, unique to each block, operates as a read-/write sparse Merkle Tree, functioning as a key-value structure similar to the `L2StateTree`. It's constructed using the Poseidon hash function. The keys utilized to position each data piece within the tree are also derived using the Poseidon hash function over specific inputs. The hash function employed for the `BlockInfoTree` follows the below signature

$$(\texttt{out}[0], \texttt{out}[1], \texttt{out}[2], \texttt{out}[3]) = \texttt{Poseidon}(\texttt{c}[0], \texttt{c}[1], \texttt{c}[2], \texttt{c}[3]; \texttt{in}[0], \texttt{in}[1], \texttt{in}[2], \texttt{in}[3], \cdots, \texttt{in}[7])$$

where $\texttt{in}[i]$ are inputs, $\texttt{c}[i]$ are the "capacity elements" and each parameter of the hash function is field element on $\mathbb{F}_p$ with $p = 2^{64} - 2^{32} + 1$.

When computing keys of block-related data, the Poseidon function is used as follows:

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

Where the parameter `SMT_KEY` is fixed to the value 7 ($\texttt{SMT\_KEY} = 7$) to compute the key of block-related data and `INDEX_BLOCK_HEADER` is used 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.

When computing keys of transaction-related data, the Poseidon function is used as follows:

$$\texttt{leafKey} = \texttt{Poseidon}(0; \texttt{txIndex}[0], \texttt{txIndex}[1], \cdots, \texttt{txIndex}[4], 0, \texttt{SMT\_KEY}, 0)$$

where notice that we hash the transaction index within the block. The parameter `SMT_KEY` takes the following values when computing the key:

- `SMT_KEY = 8`: for the transaction data hash.
- `SMT_KEY = 9`: for the transaction status.
- `SMT_KEY = 10`: for the transaction cumulative gas used.
- `SMT_KEY = 11`: for the transaction logs.
- `SMT_KEY = 12`: for the transaction effective percentage.

Finally, since we can have multiple logs per transaction, the key for logs is computed as follows:

$$\texttt{leafKey} = \texttt{Poseidon}(\texttt{logIndex}; \texttt{txIndex}[0], \texttt{txIndex}[1], \cdots, \texttt{txIndex}[4], 0, \texttt{SMT\_KEY}, 0).$$

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

6.2 Processing L2 Blocks

To explain the processing of L2 blocks in fork-etrog, we will use the diagram presented in Figure 10.

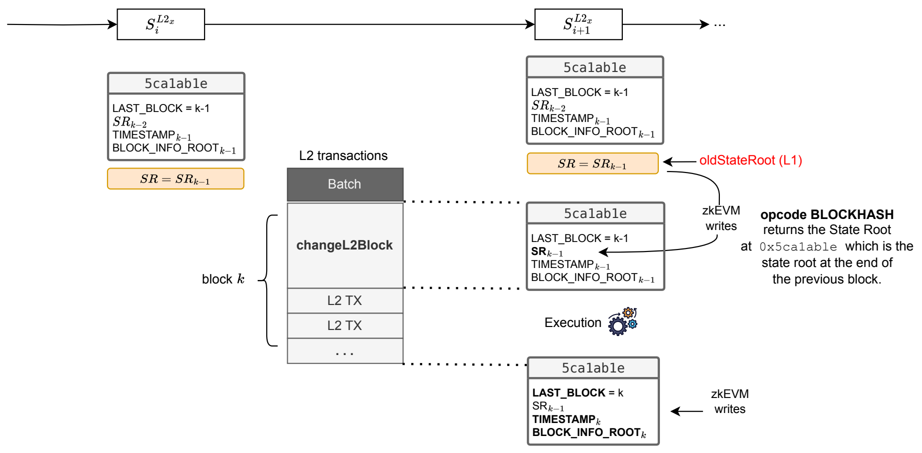

Figure 10: .

At the end of the execution of a block, the zkEVM writes the last block number, the timestamp and the block info root but **does not write the state root**. So, if we finish the execution of the block $k-1$, the parameters would be

$$\begin{gathered} \texttt{LAST\_BLOCK} = k-1, \\ SR = SR_{k-2}, \\ \texttt{TIMESTAMP}_{k-1}, \\ \texttt{BLOCK\_INFO\_ROOT}_{k-1}, \end{gathered}$$

as shown in Figure 10.

We start now processing a new block, which starts with the `changeL2Block` transaction. This transaction is always the first one to be processed in the block. The proving system provides the `oldStateRoot` $= SR_{k-1}$ and the initial step of the block's processing done the ROM is to record it in the `0x5ca1ab1e` smart contract. Afterwards, every transaction containing the `BLOCKHASH` opcode will provide the correct state root, in contrast to what happened in fork-dragonfruit. As shown in Figure 11, as transactions within the block are processed, the zkEVM not only updates the L2 state but also adds the information to the `BlockInfoTree`. Upon completing the execution of the block, we write the root of this tree into `0x5ca1ab1e`. Upon executing all transactions within the block, the `0x5ca1ab1e` contract is updated again and the whole process is repeated.

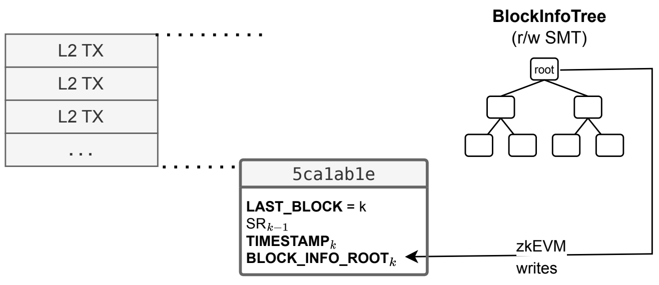

Figure 11: The `BlockInfoTree` is build while processing each transaction within the block.

It is important to mention that the state roots of the block info trees are linked, as depicted in Figure 12. The state root of the `BlockInfoTree`$_{k-1}$ is stored in the `BlockInfoTree`$_k$. The purpose of this is that we can prove any previous data since we have all the history, which can be proved using Merkle proofs.

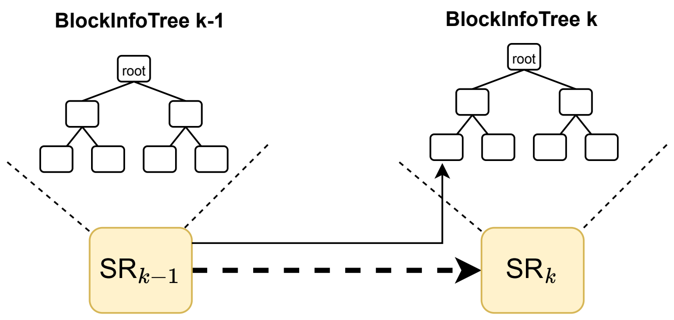

Figure 12: The state root of the `BlockInfoTree`$_{k-1}$ is stored in the `BlockInfoTree`$_k$.