zkEVM: State Management

Internal document.
Authors: Alex Belling.

The present document aims at giving a detailed view of the design of the state management. It includes the specification of a secure cryptographic accumulator and a specification of the state-manager.

The role of the state manager is to provide the Merkle proofs for access to the state (read and write), at the beginning and at the end of a block. It receives blocks created by the sequencer (technically a client node in the network), it updates the state accordingly and generates files containing the state changes with the Merkle proofs. These files are used by the provers.

The state is represented differently in a zkEVM type 2 for efficiency reasons. It means we have a different tree (SMT vs. PMT), a different way to store the values in the tree (Key Value vs. a mechanism working like a linked list), and a different hash function (MiMC vs. Keccak256).

The proposed solution must maintain two states. One in the form of PMT (Patricia Merkle trie) close to the Ethereum mainnet and which will be used to process the blocks and maintain the consensus. And another SMT trie (sparse Merkle trie) which will be used for the proof generation. It is not possible to use SMT for the consensus because the algorithm used for the hash computation is too slow and could be a problem for the block creation speed

Some design considerations:

- It should be possible to replace the hash function. Not as a configuration parameter, but as a limited impact on the code change;
- It should be possible to replace the curves used (typically changing bn254 to Goldilocks);
- The hash function is typically slow, requiring calculating the hashes in parallel;
- The targeted performances are (1) medium-term (about a year) one block of 30M gas every 3 seconds; (2) short-term, 30M gas every 12 seconds.
- If it is necessary, it is possible to require the state manager to run on a big machine such as 400 GB of memory and 96 cores (i.e., a hpc6a on AWS). The cheaper the machine the better, but it can also be a medium-term target.

Custom Accumulator based on SMT

Introduction

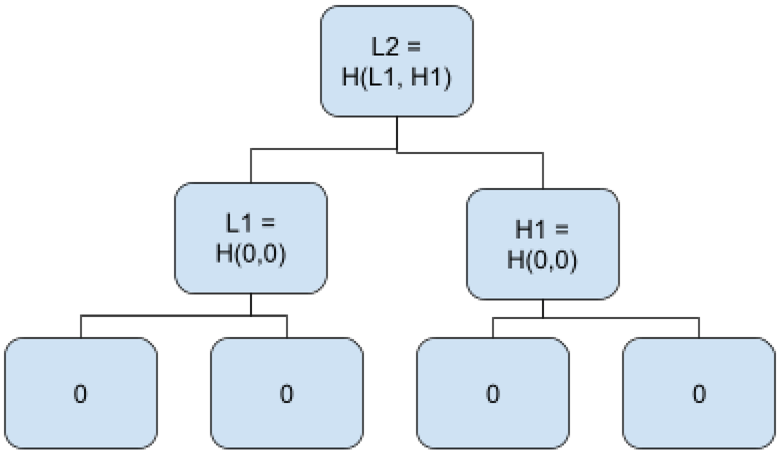
The accumulator is a data structure used in the state manager of the zkEVM to efficiently track updates to the storage of Ethereum accounts. It operates on a Sparse Merkle Tree (SMT), where each leaf represents a storage slot, and nodes represent the hash of their children. The accumulator can perform the following operations:

- **Insertion**: adding a new storage slot to the tree, triggered by storing a non-zero value in a previously zero-valued slot.
- **Update**: changing the value of an existing storage slot in the tree, triggered by storing a new non-zero value in a previously non-zero slot.
- **Deletion**: Removing a storage slot from the tree, triggered by storing the zero value in a previously non-zero slot.
- **Read Zero**: Proving non-membership, triggered when a storage slot has been accessed, but not updated, and its value is zero.
- **Read non-zero**: Proving membership, triggered when a storage slot has been accessed, but not updated and its value is non-zero.

Sparse Merkle Tree

Before diving into the complete cryptographic accumulator, we explain first what is a "sparse Merkle tree" and what we mean exactly by it.

A sparse Merkle tree is a variation of a standard Merkle tree where not all leaf nodes are filled with data. It is a complete tree of fixed depth, meaning that all branches of the tree have the same length. At initialization, all leaf nodes are set to a default value, which is typically a hash of a specific value such as zero. Because all leaf nodes have the same hash value, the parent nodes and higher-level nodes also have the same hash value. By convention, a node whose hash is the default value for its level is considered to represent an empty subtree. This allows for more efficient storage and processing of the tree, as it only needs to include the data that is present, rather than including empty leaf nodes. With this construction, we do not need to keep track of every individual node's hash, only the ones that correspond to non-empty subtrees.



Description of the cryptographic accumulator

Informally, a sparse Merkle tree can be viewed as a cryptographic accumulator that allows tracking updates of a large pre-allocated "vector" (e.g a data structure mapping integer indices to leaves). We describe a method for turning a sparse Merkle tree into a more powerful cryptographic accumulator that supports key-value addressing, rather than position addressing as in a vector. The method involves instantiating a sorted doubly linked list, where each leaf "points" to the leaves whose keys are just-above and just below in the set.

Choice of the hash functions and SMT

We use MiMC over the scalar field of the bn254 curve.

- The arity of the tree (e.g, the number of children for each node) : ***2***
- The depth of the tree: ***40***

Reference implementation

https://github.com/Consensys/gnark-crypto/blob/v0.12.1/ecc/bls12-377/fr/mimc/mimc.go
(Version 0.12.1)

Details

We use the *MiMC* hash function with Miyaguchi-Preneel construction,

- MiMC construction : https://eprint.iacr.org/2016/492.pdf
  - Using *62 rounds* for the cipher
  - And use “e = 17”, the exponent of the S-box
  - In non-Feistel Mode
- Together with the *Miyaguchi-Preneel* construction :
  - https://en.wikipedia.org/wiki/One-way_compression_function#Miyaguchi.E2.80.93Preneel
  - Where “g(x) = x” (e.g., ignore the “g” box and consider that the arrow “goes through”)
  - Where the XORs are replaced by field addition
- On the *bls12-377 scalar field*
  - *The modulus of the field is 8444461749428370424248824938781546531375899335154063827935233455917409239041*

Structure of the Merkle-tree

Leaf structure and opening of a leaf

Here, we present how to derive the value (and the implicit structure) of the non-empty leaves of the sparse Merkle tree. We set the (x) leaves as the hash of the following structure:

- **HKey**: the hash of the key (256 bits)
- **Value**: the value we wish to store (256 bits)
- **Prev**: the position of the leaf whose associated “Key” is immediately below “key” (u64)
- **Next**: the position of the leaf whose associated “Key” is immediately above “Key” (u64)

We refer to this structure as the **opening** of a leaf. Empty leaves have (cryptographically speaking) no opening since it is infeasible to find a preimage for 0.

Precise formatting for the hashing:

- All fields should be separately laid out on a separate u256 words even if the full word is not used.
- So the hashed string should be **u256(Prev) || u256(Next) || HKey || Value**

Tracking the empty leaves’ positions

Why?

Without going into the details here, when we insert into an empty storage address, the state-manager picks an empty leaf to write over. However, we have a requirement that the position in which we insert new leaves must be deterministic. Any person who looks at the transaction history should be able to reconstitute the Merkle-tree *identically* (e.g., with the same leaves at the same positions).

If the state manager was authorized to insert in any position in the SMT, then he would be able to perform an attack where he uses a random node for insertion. The accumulator state and update would still be correct, but it would be infeasible to reconstruct exactly the same SMT since the exact positions where the insertion occurs are ultimately private.

Countermeasure

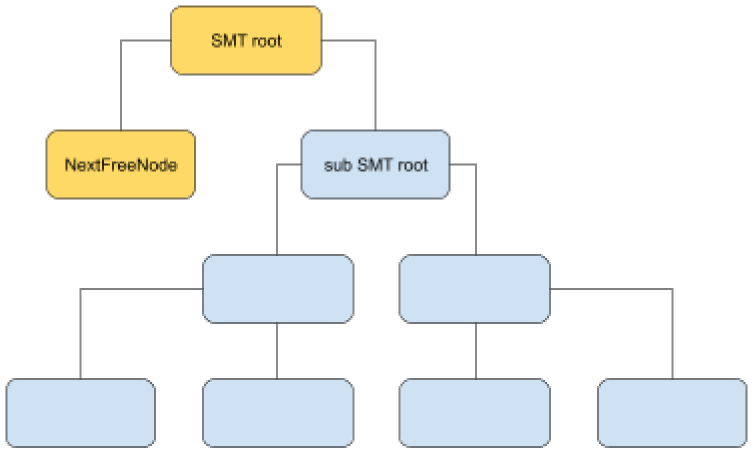
As a countermeasure, we introduce the following mechanism: we enforce the state-manager to insert to the left. Namely, the state-manager never reuses a position twice even if the corresponding node has been deleted.
=
While it may seem wasteful to not reuse erased leaves it turns out that we will never run out of leaves to write over. Under the following hypothesis (corresponding to our current L2 settings) that,

- Block gas limit = 10M
- Block time = 3s
- Inserting an account costs at least 21k and a storage slot 20k gas

Then, we would not run out of empty leaves before hundreds of years.

That being said, it should be verifiable externally without maintaining more than a root hash. For that purpose, we extend the SMT with an extra level on the top whose children are the “SMT root hash" and position of the next free node: **NextFreeNode**. Note that the depth of the tree is now 41.

Every time we make an insertion, we fetch the `**NextFreeNode**` value from the Merkle tree and increase it by 1. All updates to this field are justified by a Merkle-Proof (even though it contains only a single element) as any other leaves. With this modification in mind, when we refer to the root of the SMT, we mean the root of the upmost node and not the root of the sub-SMT root.



More precisely, the SMT root is obtained as the hash of **u256(NextFreeNode) || sub-SMT-root**

Operating the cryptographic accumulator

Initializing the cryptographic accumulator

To initialize our accumulator, we insert two ***utility leaves*** in the tree. These leaves do not contain functional data by themselves. They are only here for convenience and practicality. Insert two nodes (that we will call ***head*** and ***tail*** by convention). By "position", we mean the index of the sparse Merkle-tree leaves. Informally, in our construction, we set up a doubly linked list and **head** and **tail** represent nodes that are respectively always at the beginning and always at the end of the list. But they are not associated with any particular "key" in the set. That's the reason their **HKey** field is not the hash of something. It also ensures that these entries' values are never updated and never read.

- At position 0 (***head***)
  - **HKey** 0 (and not Hash(0))
  - **Value** 0
  - **Prev** 0 // It points to itself (invariant)

    - **Next** 1 // Points to the tail at the beginning, but this will be updated as we add entries in the set

- At position 1 (***tail***)
    - **HKey** = “`0x12ab655e9a2ca55660b44d1e5c37b00159aa76fed00000010a11800000000000`”
      (formatted as a big-endian big integer on 256 bits) (and not its hash)
    - **Value** 0
    - **Prev** 0 // Points to the head at the beginning, but this will be updated as we insert entries in the set
    - **Next** 1 // Points to itself (invariant)

INSERT (Proof and Verification)

To insert an entry **(k, v)** in the map, we need to find the two “surrounding” keys **HKey-** and **HKey+** in the list. More precisely, **HKey-** and **HKey+** are (respectively), the largest (smallest) keys such that **HKey- < hash(k) < HKey+.** Since our structure is initialized with **head** and **tail**, we are guaranteed that these entries exist. We note **i-** and **i+** the position of their respective leaves in the tree. It is possible to use an indexing mechanism (hashmap, btree, …) to find the position of a node corresponding to a key **k**. We leave this as an implementation detail.

- Find **i = nextFreeNode**, as described in the section *tracking empty leaves* the position of an empty node in the set. We expand later on the precise mechanism to derive **i.** All we say at this point is that **i** is obtained by a deterministic and verifiable procedure and **i** is the position of an empty leaf in the set.
- Get the Merkle-proofs for the leaves **i+, i-** and **i**. And the openings nodes for **i+** and **i- (N+** and **N-)**
- Normally, we have that **N+.Prev == i-** and **N-.Next == i+**. By checking this, the verifier knows that there is no intermediate key **HKey’** such that **HKey- < HKey’ < HKey+**
- Update the sparse Merkle tree as follows:
    - Get the Merkle proof of **N-**
    - Update **N-** by setting **N-.Next == i**
    - Update the Merkle-tree by inserting the new value of **hash(N-)**
    - Get the Merkle proof for the node at position **i**
    - Instantiate **N = (hash(k), v, i-, i+)**
    - Update the Merkle-tree by inserting the new value of **hash(N)**
    - Get the Merkle proof of **N+**
    - Update **N+** by setting **N+.Prev == i**
    - Update the Merkle-tree by inserting the new value of **hash(N+)**
    - Get the Merkle proof for the **nextFreeNode**
    - Update the SMT with **nextFreeNode++**
    - Derives the new root hash from the Merkle proof
- The output of the prover is
    - The three Merkle-proofs
    - The Merkle-proof for “**i == nextFreeNode**”

    - The openings **N-** and **N+** before modification
    - The positions **i+** and **i-, i**

In total, we need to update three nodes in the Merkle proof. It is important to fetch a Merkle-proof and update one leaf at a time.

The verification process goes as follows. The proof of insertion contains

- Check that **N+.Prev == i-** and **N-.Next == i+** and **HKey- < hash(k) < HKey+**
- Check that the alleged value of **N-** is consistent with the provided Merkle-proof and the current root hash
- Derive an updated root hash by updating the new value of **N-** (this can be done by reusing the same Merkle-proof)
- Check that the updated root hash is consistent with the Merkle-proof for position **i** and the leaf **0**
- Derive a (once again) updated root hash by verifying the same Merkle proof with the new inserted leaf **hash(N)**.
- Check that the alleged value of **N-** is consistent with the provided Merkle-proof and the current root hash
- Derive an updated root hash by updating the new value of **N-** (this can be done by reusing the same Merkle-proof)
- Likewise, check that the alleged value of **nextFreeNode** is consistent with the root. And derive the new root hash from it using the **nextFreeNodeMerkleProof**

READ ZERO

Let's say, we want to convince an external verifier that a key **k** is not part of the map. The process is analogous to the insertion process. For this reason, we provide a shorter description of the process. We refer to the above section for more details.

- Find **k-** and **k+**, the 2 surrounding keys and their positions **i-** and **i+**
- Send the openings **N-** and **N+**, the openings and their respective Merkle proofs

The verification process:

- Checks the Merkle proof against the alleged openings and the current root hash
- Check that **N+.Prev == i-** and **N-.Next == i+** and **HKey- < hash(k) < HKey+**

DELETE a key

It is the same idea as for removing an item from a doubly linked list. And it is in essence the reverse operation of the insertion. For more details, we refer to the "insertion" section.

Proving process:

- Find **k+**, **k-**, **i+**, **i-** and **i**
- Get the openings **N-, N, N+**
- Get the Merkle proof for **N-**

- Update the value of **N-**, by making it point to **N+** instead of **N** in the sparse Merkle tree
- Update the rest of the sparse Merkle tree
- Get the Merkle proof for **N**
- Replace the leaf **hash(N)** by **0**
- Update the rest of the sparse Merkle tree
- Get the Merkle proof for **N+**
- Update the value of **N+** by making it point to **N-** instead of **N** in the sparse Merkle tree
- Update the rest of the sparse Merkle tree
- And **i** to the list of free nodes

As for the insertion the proof of removal consists of
- The three Merkle-proofs
- The openings **N-**, **N** and **N+** **before** modification
- The positions **i+, i** and **i-**

And the verification process follows naturally

UPDATE a value

This part is identical to updating and proving the update of a value in a normal Merkle-tree

Proving process
- Find **i** and **N**
- Get the Merkle proof for **N**
- Return the Merkle proof, **N** and **i**

Verification process
- Check the Merkle proof for **N** against the current root hash
- Get the new value of **N** by changing its “Value” field
- Derive the new root hash using the Merkle proof and the new value of **hash(N)**

READ NON ZERO (or just **READ**)

This is a normal Merkle proof verification

Proving process
- Find **i** and **N**
- Get the Merkle proof for **N**
- Return the Merkle proof, **N** and **i**

Verification process
- Check the Merkle proof for **N** against the current root hash

The state manager

In the global architecture

See the architecture document for reference

Concrete usage of the accumulator and MiMC

Here, we develop how the cryptographic accumulator is used to authenticate the state concretely. The EVM naturally relies on Patricia Merkle Tree on the following points,

- World state (Accumulating the accounts)
- Account storage state (whose root is implicitly contained in the world state for each account)

For each of these, we design an equivalent structure based on the custom accumulator that we described above.

The world state (or account trie)

To represent the World State, we use our custom Merkle-tree construction. It is addressed by the account addresses and the "value" are the hashes of the account tuples similar to the EVM.

- **HKey** : Hash(address)
- **Val** : Hash(nonce, balance, storageRoot, codeHash, keccakCodeHash, CodeSize)

For the formatting of the value. The hash function that we use intrinsically relies on a finite field interpretation of the data to hash. This point is critical on the prover side. It is critical that the hashed data are formatted in a way that is consistent with this finite-field interpretation. We propose the following:

- 1 field element for the nonce.

    - The nonce is written in big endian form into a byte32. For instance if the nonce is 10. Then the nonce should be encoded as `0x000000000000000000000000000000000000000000000000000000000000000a`

- 1 field element for the balance
    - Same as the nonce

- 1 field element for the storage root. T**he storage root should not be the keccak Patricia trie root as in the EVM** but the "Custom Merkle Tree" root of the account storage state that we describe in the following section.

- 1 field element for the codeHash. **The code hash should not be the keccak of the code**, it should instead be the one obtained as described in the following section.
- 2 field elements for the **keccakCodeRoot** hash. 1 one for the 128 most significant bits and 1 for the 128 least significant bits. The keccak code hash corresponds exactly to the keccak hash as specified by the EVM (e.g., the output of EXTCODEHASH. We keep the keccak and the "custom" version for practical reasons.
- 1 field element for the **codeSize** (it should be the same value as what is returned by the CODESIZE/EXTCODESIZE opcodes)

The account storage state (or storage trie)

The account storage state is represented as a

- **HKey** : Hash(StorageKeyMSB, StorageKeyLSB)
- **Val**: Hash(StoredValueMSB, StorageValLSB)Y

Since the storage key and values can be any arbitrary number between 0 and 2^256 and that the finite field has an order of 2^254, we need 2 field elements to describe a whole word. The MSB takes the 16 first bytes of the word and the LSB the 16 last bytes. (Implicitly, we assume bigendianness)

Example:

If I have the following bytes32 : `[a0, a1, a2, …., a15, b0, b1, …, b15]`

Then, I should have the resulting MSB, LSB

```
MSB = [0, 0, .., 0, a0, a1, a2, a3, .., a15]
LSB = [0, 0, .., 0, b0, b1, b2, b3, .., b15]
```

The code hash

The code of a contract is hashed in two ways : Keccak and MiMC. In this part, we explain how the code should be passed to the MiMC hasher precisely. Since the MiMC hasher operates over field elements and the overall operation should be ZK friendly we cannot just say that we pass a string of bytes because that would have many overheads on the prover.

Each opcode making up the code to hash fits on a single byte. Since it would be too inefficient to use one field element per opcode we group them in "limbs" of 16 bytes (so 16 opcodes per limbs).

limb = opcode_0 || opcode_1 || opcode_2 || … || opcode_15)

Operating the accumulator in the state-manager

As a result of optimization in the zkEVM, the state-manager only needs to operate the accumulator once for each accessed/updated memory slot. In other words, the state accesses are cached. This means that it only matters to look at the initial and final value of an accessed account to determine the type of operation that should be performed.

The updates are done in the following order,

- Update all the accessed account storage slots
- Update all the accounts in the world state

The order of the storage accumulator operation is not relevant by itself, but:

- All operations should be done sequentially (or it should look as if it was done sequentially from looking at the output)
- It should be tractable for the “prover” to understand in which order he has to audit the storage accesses (remember that these are sequential operations over a Merkle-tree)

Sequencing the Merkle proofs updates

Namely, the update workflow can be summarized with the following pseudo-code. It intends to cover all cases.

```
// Update the world-state account by account
for address := range touchedAddresses {

    // We distinguish several cases according to the
    // following flags. This flags are all compatible
    // with each other.

    // Flag indicating that the account is in a
    // "non-existent" state at the end of the block.
    // This can happen if SELFDESTRUCT was called during
    // the execution
    var accountDestructed bool

    // Flag indicating that the account was non-existent
    // at the beginning of the block execution.
    var accountCreated bool

    // Flag indicating that the contract has had several
    // deployments during its execution.
    var accountRedeployed bool
```

```
// Trivial case : the account is non-existent at both
// the beginning and the end of the block execution. In
// this case, we do not need to update the state. NB:
// this edge-case is explicitly written for the sake of
// clarify but all the cases below exclude it already.
if accountCreated && accountDestructed {
    continue
}

// Regular case : the account already exists, still
// exists at the end and has not been redeployed
if !accountCreated && !accountDestructed && !accountRedeployed {

    // For all accessed slots, detect the type of access
    // that we need to perform (READ_NON_ZERO, READ,
    // UPDATE, DELETE, INSERT) from looking at the
    // initial and the final value. IGNORE ALL REVERTED
    // ACCESSES.
    for stateDelta := range accessedSlots {
        access_with_trace(stateDelta)
    }

    // Once we have traced all the storage accesses
    // up to the final STORAGE_ROOT_HASH value. We can
    // update the account in the world state
    access_with_trace(address, accountDelta)
}

// Edge-case land : the contract has interacted with
// either CREATE, CREATE2 or SELF-DESTRUCT in its life.
// The two following clauses summarize the process
// for all combination of SELFDDESTRUCT/CREATE2

if (accountRedeployed || accountDestructed) && !accountCreated {

    // 1/ Only look at the initial value of each
    // accessed storage slot BEFORE THE FIRST NON-
    // REVERTED SELFDESTRUCT. And only generate
```

```
            // traces for the READ/READ_ZERO for the initial
            // values. NB, we are guaranteed that a
            // SELFDESTRUCT occurred with the condition that
            // we have set on the flags.
            for slot := range accessesBeforeFirstSelfdestruct {
                read_with_trace(slot, initialValue)
            }

            // 2/ DELETE the account from the world-state
            // and generate a trace of the deletion.
            delete_with_trace(address)
        }

        if (accountRedeployed || accountCreated) && !accountDestructed {

            // 3/ Only look at the final values of any accesses
            // storage slot AFTER THE LAST NON-REVERTED CREATE/
            // CREATE2. And generate traces for the insertions
            // of the last values.
            for slot := range accessesAfterLastCreate {
                insert_with_trace(slot, finalValue)
            }

            // 4/ INSERT a new account (as a result of the LAST
            // CREATE2) implicitly this account will have the
            // same address as the one we deleted but it will
            // "live" in another slot of the tree. All its field
            // corresponds to their final values after the
            // execution.
            insert_with_trace(address, accountFinalValue)
        }
    }
```

The advantage of that approach is that it allows us to compute only once the Merkle proof of each account per update.

Case by case examples: Account Storage

The section below clarifies which accumulator operation to use when a storage slot is updated.

Insertion

If the initial state X is 0 and we have the following sequence of instructions

- `SSTORE X 1`
- `SLOAD X`
- `SSTORE X 5`
- `SSTORE X 6`

Then, in that context, the state-manager should only perform “**Insert X 6**” at the end of the block, ignoring the intermediate state, and collect the proof of insertion. We recall that storing the word zero amounts to storing nothing. That’s why it is an assertion and not an update.

This should also be triggered in the world state when an account is created.

Update

If the initial state for X is 2. And we have a block containing the following operations on X

- `SSTORE X 2`
- `SLOAD X`
- `SSTORE X 6`

In that context, the state manager performs “**UPDATE X 6**”. This should also happen in the world state when an account is updated.

Deletion

If the initial state is for X is 2, and we have the following sequence of instruction in a block

- `SSTORE X 3`
- `SSTORE X 0`

In that context, the state-manager performs a **DELETE X**

This should also happen in the world state when an account is destroyed. As we understand (zkEVM team), an account can be either in “dead” mode or in “unlisted mode”. But at the end of the block, all dead accounts are unlisted, so the state-manager does not have to do this distinction (aside its vanilla EVM operations).

Read Zero

If the initial state is zero, and we don’t write in X **OR** the initial state is zero the last write in X is the zero word 0. Then, the state-manager proves non-membership of X (e.g., that X is missing from the set)

For instance, the two following sequences of operations trigger a proof of non-membership

• `SLOAD X` • `SLOAD X`	• `SSTORE X 2` • `SLOAD X` • `SSTORE X 0` • `SSLOAD X`

Read non-zero

If the initial state is non-zero and we only perform a **READ (READ NON ZERO)** operation. Then, we only ought to assess that the key is present. For instance, if X is initialized to 2 and we have the following sequence of instructions

- `SLOAD X`
- `SLOAD X`

Then, the state manager should perform a **READ X** operation at the end of the block. As above, if the storage slot is modified but is set back to its initial value during the operation, then it is also a **READ NON ZERO.**

There is a REVERT somewhere

While this may seem like a special case. It is captured by the idea that we only care about the initial value of a storage slot or an account.

If the account storage slot is written to by only a single transaction that is later reverted, then the **SSTOREs** that occurred during the execution are reverted. Namely, all states write operations done within a reverted transaction count as a **READ** because the zkEVM has to justify how the transaction got reverted. For instance,

Initially, **Val[X] = 3**
============
Tx 1

- `SLOAD X`
- `SSTORE X 2`
- `REVERT`

Then the state manager ought to perform a **READ NON ZERO**. If the initial value was Val[X] = 0, it would have been a **READ ZERO.**

If the block execution accesses a storage slot several times across several transactions and one (or several of them) are reverted, we treat it as if there were **SLOADs** in these transactions.

For instance,

Initially, **Val[X] = 3**
============

Tx1

- `SLOAD X`
- `SSTORE X 2`
- `REVERT`

============

Tx2

- `SSTORE X  5`

============

Tx3

- `SSTORE X 6`
- `REVERT`

In this example, Tx2 is sandwiched by 2 reverted transactions. The resulting accumulator operation for X is **UPDATE X 5**.

Updating an account

The account trie update mechanism follows the same logic as the storage tries. We determine the accumulator operation to perform by looking at the value of the account before and after the block execution. By value of an account, we mean the tuple **(balance, nonce, storageHash, codeHash, codeHashLegacy)**

- If the account's value is the same at the beginning and at the end, and the value is at least accessed. Then it's a **READ** if it exists or **READ ZERO** if either nonexistent or created-deleted in the same block.
- If the account is created, then we **INSERT** the final value
- If the account initially exists and was deleted, then it's a **DELETE**
- If the account exists and its value is updated, then it's an **UPDATE**.

Operations involving the Code-Hash

Initializing/Updating the code hash of an account

When this happens, we update the corresponding account with

- A new keccak code hash (computed as specified in the yellow paper when returning from `EXTCODEHASH`)
- The codehash obtained by hashing the code using MiMC
- The code size as specified in the yellow paper as the return argument of `EXTCODESIZE/CODESIZE`
- Account the overall operation as an **UPDATE/INSERT** of the account (depending on the case)

Reading the code of an account

When we load the code of a contract (as a result of a `CALL/DELEGATECALL/CALLCODE/STATICCALL` etc..) or call the opcode `EXTCODECOPY/CODECOPY` (non-exhaustive), the state-manager authenticates the “loaded” code.

- By a **READ** of the account (e.g justify the value of the account)
- Sending the codehash explicitly to the prover

Deleting the code of an account

Unless mistaken, this cannot happen without deleting the account
Example : Ordering the state operations

Let's say that we have a block of transactions. The state manager operates as follows

1) **Execution**

We execute the block’s transaction without touching the custom accumulator. During the execution, we keep track of the state slots (accounts and account storage) that were touched. Now, let assume that throughout the execution of the block, the only account that has been accessed is “A” and that we have the following accesses to its account storage.

SLOAD X (=0)
SSTORE X 1
SLOAD Y (=5)
SSTORE Y 2 (this one will be the proof after X insertion in the trie)
SLOAD X (=1)
SSTORE X 2 (this one will be the proof after Y insertion in the trie)

This being recorded, we sum up the state delta for the account storage of A. As suggested in the table below. We have two operations to run on the accumulator for the account storage of A.

Slot	Init	End	Operation
X	0	2	INSERTION
Y	5	2	UPDATE

2) **Operating the account storage accumulator for A**

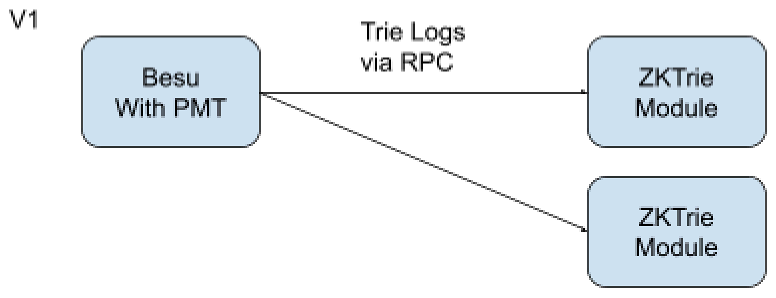
The operations on the accumulator are run “key by key” starting with the smallest key and finishing with the largest key. The order in itself is not relevant, but should be deterministic or inferable by the prover. Let's say that X goes before Y.

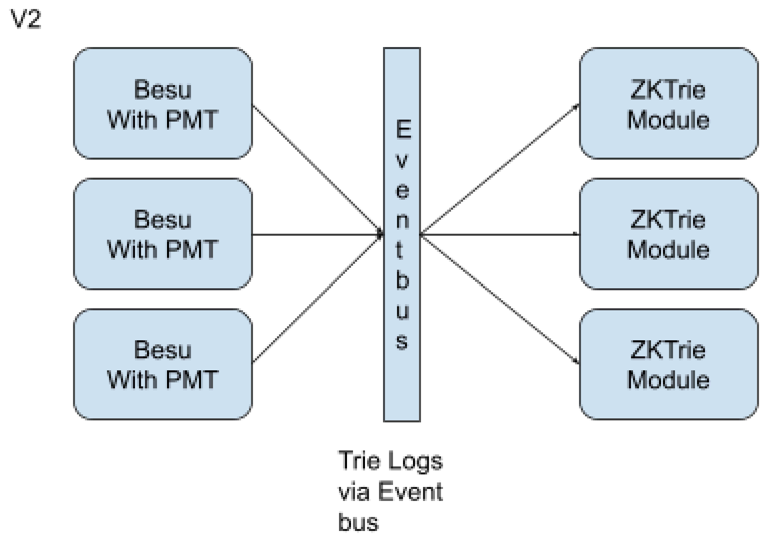
X	It an INSERT, so we follow the specified process for insertions ● Find the nextEmptySlot ● Find i+ and i- the positions of the LeafOpenings such that : HKey- < Hash(XMSB, XLSB) < HKey+. We will update them as specified in the accumulator section ● Generate the proof for the leaf i- ● Update the tree ● Generate the proof for the leaf ‘nextEmptyProof’ ● Update the tree ● Generate the proof for i+ ● Update the tree
Y	It’s an UPDATE, so we follow the specified process ● Find i, the position of the leaf whose HKEY is H(YMSB, YLSB) ● Generate the proof for i ● Update the “V” part of the LeafOpening ● Update the tree with the new leaf

Format of the output of the state-manager

See the API spec document for reference

Two Tries system:





We will separate the PMT and ZKT into two modules. Besu will not change and will work on Bonsai in order to maintain the PMT state, import block etc.

The ZKEVM component is standalone and is not in Besu. It will contain its own database, its own RPC module and its own state. you should have to call it to retrieve the proof. This component will be very light and will have a very small scope : Update State and generate proof. The component will be able to generate proofs of an old state thanks to the trielog which can allow it to find a state of a block n. And can also generate the proof of several blocks at the same time

When Besu imports a new block it will send the trielogs to the ZKT module. ZKT module will be able to understand Bonsai trielogs from Besu (this will require a small trielog adaptation -> adding zero read). ZKT module will also have the ability to request a trielog from Besu if it is behind Besu's head . For example Besu sends the trielog of block 4 but the head of the ZKT module is 2. So it will therefore request block 3 trielog

The V1 will use RPC for the communication between the two components . This will add the ability to duplicate the proof generation component. A single Besu can manage several ZKT components

The V2 will use an event bus for the communication. This will add the ability to duplicate Besu and also the ZKT module in order to remove the possibility of a single point of failure