polygon zkEVM

Knowledge Layer

Specs

L2 State Storage

v.1.0

January 11, 2024

1 A zk-Friendly Sparse Merkle Trie for the zkEVM State

The Merkle tree we are using to store the zkEVM State is a Sparse Merkle Binary Trie fully updatable (that is, allowing both read and write operations) used to represent key-value data. We will also make it radix in order to optimize storage space. Due to its construction, it will be unbalanced by nature (since keys, which are used to completely locate leaves within the tree, will be hashes of some data).

Underlying Hash Function `Poseidon` is a cryptographic hash function that has garnered attention for its unique properties and suitability in the context of Zero-Knowledge proofs. It was originally designed to address some of the limitations and challenges faced by traditional hash functions in this specific application domain. From now on, we will consider an instantiation of `Poseidon` over a Goldilocks field (that is, $\mathbb{F}_p$ where $p = 2^{64} - 2^{32} + 1$), that takes as entry 8 field elements divided into two groups: four of them are know as **capacity elements**, and the other 4 are the elements to be hashed, known as **input elements** and outputs 4 field elements known as **output elements**. We will denote a Poseidon hash as follows:

$$\texttt{Poseidon}(\texttt{capacity}; \texttt{values})$$

Here, `capacity` is an array of 4 field elements representing the capacity elements, and `values` is an array of 4 field elements representing the inputs of the hash. Observe that we can also see `values` as an array of 8 (almost) 32-bits elements. In our context, two instances of Poseidon hash function are employed for protecting the tree against second-preimage attacks: $\texttt{Poseidon}(0; \cdot)$ for branch nodes and $\texttt{Poseidon}(1; \cdot)$ for leaf nodes.

A Glimpse Through Toy Example Let's visualize the tree using an example. Lets suppose we have some set of data associated with four keys: `000110`, `001011`, `001100`, and `101101`. Prefixes are used to position the leaves containing data. We have four distinct prefixes: `000`, `0010`, `0011`, and `1`. Recall that zero nodes are used to signify the absence of keys under a given prefix. With this design, similar keys share a common path in the tree, saving space.

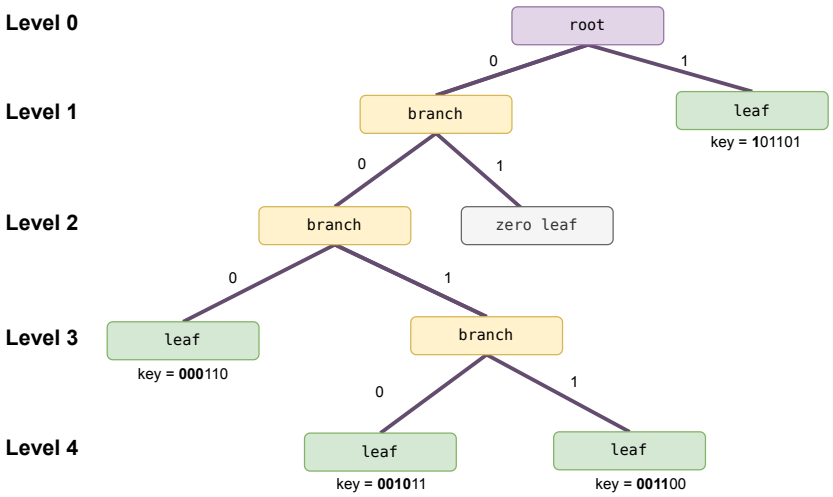

Figure 1: Illustrative Example of a Sparse Merkle Trie.

The Four Node Types In the proposed example we can see the 4 different types of nodes appearing in our tree design:

- **Root node**: The root node is the top node of the tree, the only one node at level zero, connecting all branches and leaves. It is the starting point for traversing the tree structure. It serves as a cryptographic summary of all the tree, the final hash of the entire dataset.
- **Leaf node**: Leaf nodes encapsulate the actual data associated with a key. Its stored value can be obtained by hashing unique information of each key-value binding of the dataset.
- **Branch node**: Branch nodes serve as intermediaries within the tree structure, positioned at various levels between the root node and the leaf nodes. These nodes actively guide the traversal path within the trie. The primary responsibility of a branch node is to store the hash derived from concatenating the hashes of its two child nodes. This hash encapsulates the segment of the cryptographic summary representing the data contained in the subtree rooted at that branch node.
- **Zero node**: Zero nodes **do not** represents a zero value for a specific key. Instead, they denote the absence of a subtree beneath a branch node. In our example, we can see that all keys starting with 01 share the default value of 0. Consequently, there is no need for these keys to be individually present as nodes in the tree, and this is where zero nodes come into play. However, the tree actually stores several keys starting with 00. These keys require the presence of a branch node at level 2 to obtain the parent's node hash. The optimization lies in the decision not to hash the subtree below with hashes of zero but rather to represent it with an explicit zero value. In practical terms, this means that the branch node at level 1 assumes a specific value:

  $$\texttt{Poseidon}(0; \texttt{leftChildNode}, 0).$$

  This optimization is called **partial tree construction**.

2 Keys and Values

In the context of managing key-value data within a Merkle tree structure, it is important to correctly specify the process by which keys and values are employed to create and position data within the tree. The leaf nodes serve as the endpoints of the tree and essentially hold data.

2.1 Obtaining the Keys

The current methodology employed for hashing the leaf data to derive the node's key unfolds as follows:

$$\texttt{leafKey} = \texttt{Poseidon}(\texttt{capacityHash}; \texttt{addr}, \texttt{SMT\_KEY}, 0)$$

The array

$$\texttt{addr} = (\texttt{addr[0]}, \texttt{addr[1]}, \texttt{addr[2]}, \texttt{addr[3]}, \texttt{addr[4]})$$

comprises a 32-bit segmented representation of the Ethereum address associated with the leaf, having 160 bits in the EVM. `SMT_KEY` is a single field element that takes the following values for the different L2 state information being stored:

- `SMT_KEY = 0`: for the account balances.
- `SMT_KEY = 1`: for the account nonces.
- `SMT_KEY = 2`: for the smart contracts' code.
- `SMT_KEY = 3`: for the smart contracts' storage.
- `SMT_KEY = 4`: for the smart contracts' length.

The computation of the capacity hash differs depending on whether we are storing a storage slot or some other type of data.

(a) When storing data that is not a storage slot, the capacity hash is computed as follows:

$$\texttt{capacityHash} = \texttt{Poseidon}(0; 0),$$

where the capacity shown as `0` is an array of 4 field elements all of them set to 0.

(b) Since a single smart contract's address can have multiple storage slots, we create different keys for each slot as follows:

$$\texttt{capacityHash} = \texttt{Poseidon}(0; \texttt{storageKey}),$$

where `storageKey` is an array of 8 field elements where each of these elements is constrained to values of 32 bits for the representation of the storage slot identifiers, which in the EVM have a size of 256-bits.

2.2 Obtaining the Values

Having specified the procedure for deriving the keys associated with the leaves, it is equally important to outline the process by which the values stored within these leaves are obtained. The current methodology employed for hashing the leaf data to derive the node's value unfolds as follows:

$$\texttt{leafValue} = \texttt{Poseidon}(1; \texttt{remLeafKey}, \texttt{valueHash}).$$

In this case, the capacity `1` is an array of 4 field elements with $\texttt{c}[0] = 1$ and $\texttt{c}[1] = \texttt{c}[2] = \texttt{c}[3] = 0$. The `remLeafKey` argument of the hash consists as an array of 4 field elements, which collectively represent the remaining key associated with the leaf node. The second argument `valueHash` is is an array of 4 field elements that is calculated as follows:

$$\texttt{valueHash} = \texttt{Poseidon}(0; \texttt{value}),$$

where `value` is an array of 8 field elements, with each element constrained to values of 32 bits. The 32-bits values in the array are computed differently depending on the type of the data intended for storage within the tree.:

- **`balances`, `nonces` and `storageSlotValues`.**

  In the EVM, `balances`, `nonces` and `storageSlotValues` are of 256 bits. So, their `valueHashes` are computed as expected:

  $$\texttt{valueHash} = \texttt{Poseidon(0; balance)},$$
  $$\texttt{valueHash} = \texttt{Poseidon(0; nonce)},$$
  $$\texttt{valueHash} = \texttt{Poseidon(0; storageSlotValue)}.$$

  All the previous arrays are of 8 field elements constrained to values of 32 bits.

- **Smart Contracts Code.**

  In the L2 state tree we also need to store the hash of the smart contracts bytecode. The issue is that the bytecode of smart contract can be larger than 256-bits (32 bytes). Actually, it can have an arbitrary size. To manage these issues:

  – We use a **padding** to complete the size of data being hashed to a multiple of a suitable size that is 56 bytes (we see why next).

  – Then, we linearly hash the resulting padded byte string.

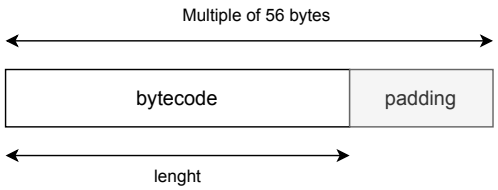

Figure 2: The padding is introduced in order to enable the bytecode to be hashed using `Poseidon`.

In the L2 state tree, we also need to store the bytecode length to distinguish what is payload from what is padding:

– The bytecode length represents the number of bytes of the bytecode.

– This is a 256-bit value.

– The `valueHash` is computed as follows:

$$\texttt{valueHash} = \texttt{Poseidon(0;bytecodeLength)}.$$

We encode the bytecode (which is multiple of a byte) in segments of 7 bytes (56 bits), which is the biggest amount of bytes that can be represented with an element of our field:

$$2^{56} < 2^{64} - 2^{32} + 1 < 2^{64}.$$

The `Poseidon` hash function signature that use has 8 inputs, so we hash in blocks of 8 field elements which encode 7 bytes $\times 8$ elements $= 56$ bytes of bytecode. So, we use a padding procedure that completes the original bytecode string to a padded string that is multiple of 56 bytes as follows:

1. Add the byte `0x01` at the end of the original bytecode.
2. Fill with zeros until the padded bytecode length + 1 is multiple of 56 bytes.
3. Add `0x80` as the last byte.

For example, if the original bytecode is `0xdead`, having size of 2 bytes, the padded bytecode becomes `0xdead01000000000000000000...000000000000080`, having size of 56 bytes.

Since smart contract's bytecode can be larger than 256 bits, we will need a chained hash process in order to obtain a valid representative value (that we will call `bytecodeHash`). More specifically, we recursively hash 8 of the 7 bytes segments along with the hash of the preceding chunk, input as the capacity element.

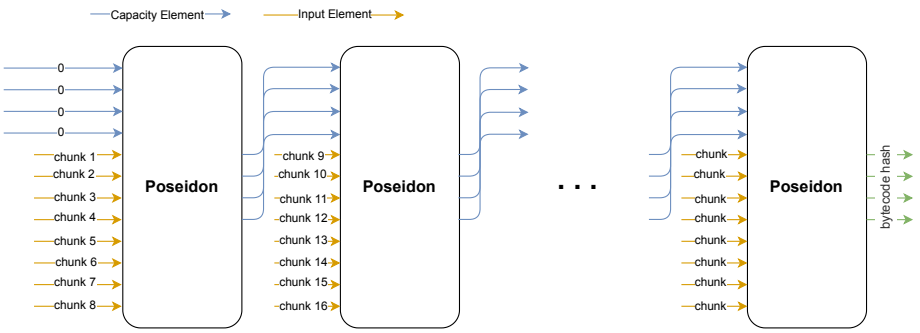

Figure 3: Notice that we use the output of the previous `Poseidon` function as the capacity in the new computation of the next `Poseidon` hash.

Following the previous example having `0xdead` as the bytecode. Recall that the padded bytecode became

`0xdead010000000000000000000000000...00000000000000000000080`.

We perform one `Poseidon` (since the padded bytecode has length 56 bytes) with the following 12 elements:

```
[ "00000000000000", (7 bytes) (element 0)
  "00000000000000", (7 bytes) (element 1)
  "00000000000000", (7 bytes) (element 2)
  "00000000000000", (7 bytes) (element 3)
  "0000000001adde"  (7 bytes) (element 4)
  "00000000000000"  (7 bytes) (element 5)
  "00000000000000"  (7 bytes) (element 6)
  "00000000000000"  (7 bytes) (element 7)
  "00000000000000"  (7 bytes) (element 8)
  "00000000000000"  (7 bytes) (element 9)
  "00000000000000"  (7 bytes) (element 10)
  "80000000000000"  (7 bytes) (element 11) ]
```

Figure 4: The `Poseidon` hash is performed with 12 elements, each representing a 7-byte value. Since we are performing the first 7 bytes, we leave the capacity to `0`.

Finally, the `valueHash` of the bytecode is `valueHash` = `Poseidon(0; bytecodeHash)`.