Sparse Merkle Trees

Jordi Baylina[1] and Marta Bellés[1,2]

[1] *iden3,* [2] *Universitat Pompeu Fabra*

Contents

1 Scope

A Merkle tree or hash tree is an authenticated data structure where every leaf node of the tree contains the cryptographic hash of a data block and every non leaf node contains the concatenated hashes of its child nodes [2]. If the majority of the leaves are empty, then they are called sparse Merkle trees [1]. This proposal aims to standardize the generation of this second kind of binary trees.

2 Motivation

Merkle trees allow to link a set of data to a unique has value, which is very optimal and useful, specially in blockchain technology, as it provides a secure and efficient verification of large data sets by storing only a little piece of data on-chain. For instance, they can be used to verify any kind of data stored, handled and transferred in and between computers. They can help ensure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and even to check that the other peers do not lie and send fake blocks [3].

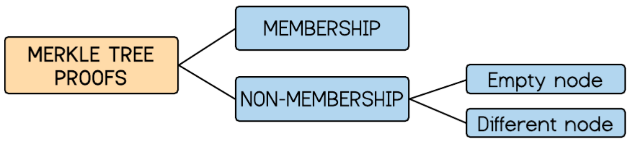
3 Background

We are still working on the literature compending the state of the art of this area.

4 Terminology

The following concepts are definitions and properties we assume across the document.

- The leaves of the *Merkle tree* consist of key-value pairs (k, v). We distinguish three different nodes:
  - *Empty node*: A vertex that stores the key and value zero.
  - *Leaf*: A vertex with both empty children.
  - *Internal node*: A vertex with at least one non-empty child. The value is and the key such. It has the hash of its children.
- A *Merkle audit path* for a leaf in a Merkle tree is the shortest list of additional nodes in the tree required to compute the root hash for that tree.
- If the root computed from the audit path matches the true root, then the audit path is a *proof of membership* for that leaf in the tree.
- Otherwise, it is a *proof of non-membership* for that leaf in the tree.



5 Challenges

Work in progress.

6 Description

Let $e = (k, v)$ be a new entry in a tree T. The node in which this piece of data e is stored in T is uniquely determined from the data itself. Let H be a secure hash function returning an array of bits [1]. The leaf in which e should be stored in T is defined by

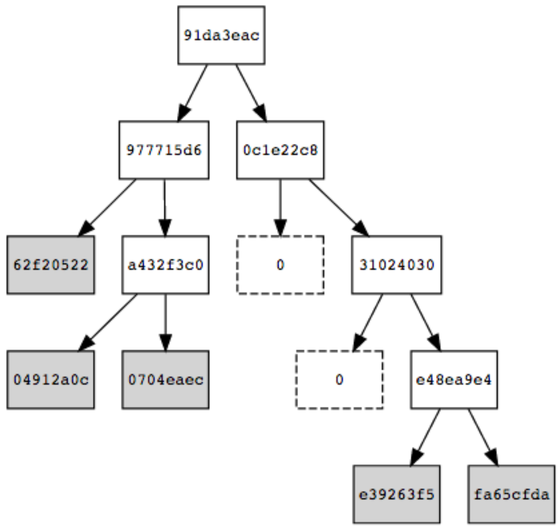
$$H_{path} = H(e) = H(1||k||v).$$

This array of bits is going to represent a path through the tree: starting by the less significant bit and from the root of T, it descents the tree by taking the left edge if there is a 0 and right one if there is a 1.

When adding an entry e, we may not (see Sec. 7) go down to the last level of the tree (by last we mean looking at all the bits, length of which depends on the hash function H). What we do instead, is go down through the path until we find a node without siblings (a leaf). If the leaf is empty, we store e. Otherwise, that node stores some other e' (as non-empty leafs store claims) with $H(e') = H'_{path}$. This means that H_{path} and H'_{path} start with the same sequence of bits. We compare both hashes and go down the tree until the first different bit. these two values and find the first different bit (included). Then we store e and e' in their corresponding leafs of the path.

Example

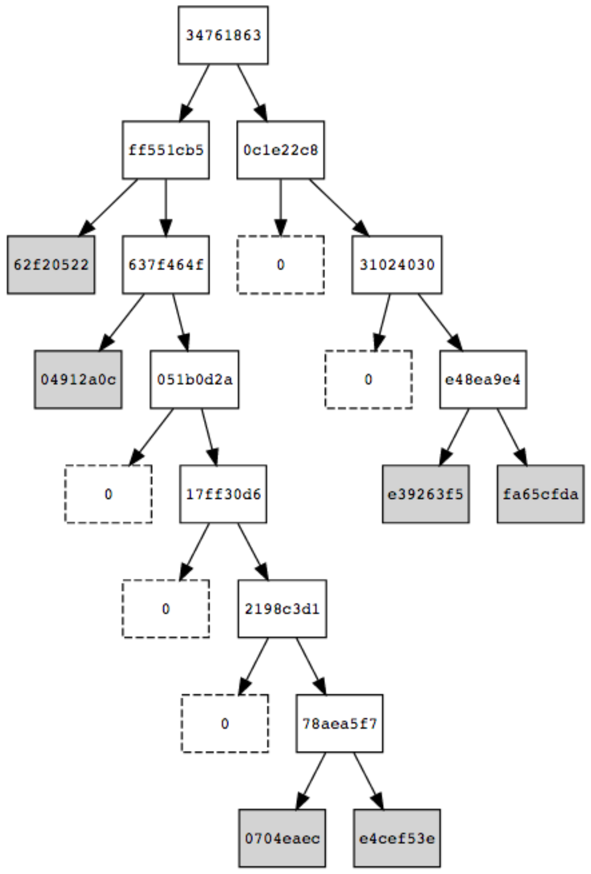
As an example, consider e such that H_{path} = 0111111... and the Merkle tree below where in each leaf there is represented the value (and not the key) of each stored piece of data:



If we go down the tree following the sequence 01111111... we get to the leaf containing the value 0704eaec of some e' with H'_{path} = 0111110... . Comparing H_{path} and H'_{path}, the 7th bit is

[1]If the hash function H does not return a binary number, binarize it later.

the first different bit. This means, that we should go down to the 7th level and store there the entries as shown in next figure:



Note that e is stored in the right (as the 7th bit is a 1) and e' is stored in the left (as it is a 0). Also note that the rest of siblings are empty nodes and how the root and intermediate nodes have changed.

Remark

Each node is of the form $(H[b, k, v])$, where $b = 1$ if terminal node (leaf) and $b = 0$ otherwise. More precisely,

- Each leaf consists of a pair $(H(1||k||v),\ k||v)$.
- Each intermediate node of a pair $(H(H_L||H_R),\ K_L||K_R)$, where (H_L, K_L) is the key-value of its left child and (H_L, K_L) the key-value of its right child.

Pseudocode

The procedure to store an entry in a Merkle tree is described below in pseudocode.

```
procedure INSERT ENTRY $e$ IN MERKLE TREE $T$ WITH ROOT $r$
    $H_{path} \leftarrow$ GetPath($e$)
    $b \leftarrow$ LeastSignificantBit($H_{Index}$)
    if $r$ is empty then $r \leftarrow e$
    else
        while $r$ is internal vertex do
            if $b = 0$ then $r \leftarrow$ LeftChild($r$)
            else $r \leftarrow$ RightChild($r$)
            if $r$ is empty then
                $r \leftarrow e$
                $H_{Index} \leftarrow H_{Index} \backslash b$
                $b \leftarrow$ LeastSignificantBit($H_{Index}$)
    $e' \leftarrow$ GetEntryStoredIn($r$)
    $H'_{path} \leftarrow$ GetPath($e'$)
    if $H_{path} \neq H'_{path}$ then
        Find first bit $b_j$ such that $H_{path}(j) \neq H'_{path}(j)$
        Leaf($b_0 ... b_j$)$\leftarrow e$
        Leaf($b_0 ... b'_j$)$\leftarrow e'$
        RecalculateIntermediateNodeValues($T$)
```

We are working on 4 more procedures: On one side, DELETE of entries and UPDATE of the tree. On the other side, the generation of MEMBERSHIP proofs and generation of NON-MEMBERSHIP proofs.

These last two procedure, although *we are working on explaining them in detail in the following delivery*, they have already been implemented in GoLang and JavaScript in the following two repositories:

- `https://github.com/iden3/go-iden3/blob/master/merkletree/`
- `https://github.com/iden3/iden3js/tree/master/src/sparse-merkle-tree`

7 Security

The security of an audit path reduces to the collision resistance of the underlying hash function. For a proof, see [1, Lemma 1].

8 Implementation

The standarisation of Merkle trees we proposed are described an implemented in GoLang and JavaScript by the iden3 team in the following repositories:

- `https://github.com/iden3/go-iden3`

- `https://github.com/iden3/iden3js`

Some detailed examples are also provided in these repositories:

- `https://github.com/iden3/go-iden3/blob/master/merkletreeDoc/merkletree.md`
- `https://github.com/iden3/iden3js/tree/master/examples`

9 Intellectual Property

We will release the final version of this proposal under creative commons, to ensure it is freely available to everyone.

References

[1] Dahlberg, R., Pulls, T., and Peeters, R. Efficient sparse merkle trees: Caching strategies and secure (non-)membership proofs. Cryptology ePrint Archive, Report 2016/683, 2016. `https://eprint.iacr.org/2016/683`.

[2] Haider, F. Compact sparse merkle trees. Cryptology ePrint Archive, Report 2018/955, 2018. `https://eprint.iacr.org/2018/955`.

[3] Wikipedians, B. *Data Structures*. PediaPress.