LeanIMT: An optimized Incremental Merkle Tree

Privacy & Scaling Explorations

June 11, 2024

1 Abstract

Contents

1 Abstract **1**

2 Introduction **3**
2.1 Motivation 3

3 Merkle Tree **3**
3.1 Incremental Merkle Tree 3
3.2 Binary Tree 3

4 LeanIMT **3**
4.1 Definition 3
4.2 Insertion 4
4.2.1 Pseudocode 6
4.3 Batch Insertion 6
4.3.1 Pseudocode 7
4.4 Update 7
4.5 Remove 8
4.6 Generate Merkle Proof 9
4.7 Verify Merkle Proof 9

5 Implementations **9**
5.1 TypeScript 9
5.2 Solidity 9

6 Benchmarks **9**

7 Conslusions **9**

2 Introduction

2.1 Motivation

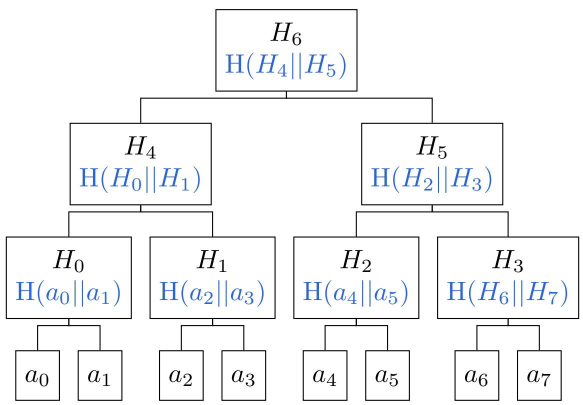
3 Merkle Tree

3.1 Incremental Merkle Tree

An Incremenental Merkle Tree (IMT) is a Merkle Tree (MT) designed to be updated efficiently.

3.2 Binary Tree

A Binary Tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child.
TODO: Explain what is a Merkle tree and an Incremental Merkle Tree.



4 LeanIMT

4.1 Definition

The **LeanIMT** (Lean Incremental Merkle Tree) is a Binary IMT.

The LeanIMT has two properties:
1. Every node with two children is the hash of its two child nodes.
2. Every node with one child has the same value as its child node.

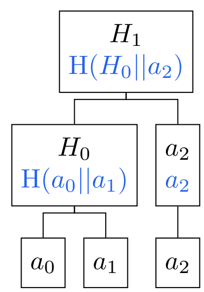
Example of a LeanIMT

T - Tree
V - Vertices (Nodes)
E - Edges (Lines connecting Nodes)

$T = (V, E)$

$V = \{a_0, a_1, a_2, H_0, H_1, H_2\}$

$E = \{(a_0, H_0), (a_1, H_0), (a_2, a_2), (H_0, H_1), (a_2, a_2)\}$



4.2 Insertion

There are two cases:
1. When the new node is a left node.
2. When the new node is a right node.

We will always see one of these cases in each level when we are inserting a node. It is like, when you insert a node, if that node is left node, the parent node which is in the next level, will be the same node. If it is a right node the parent node, will be the hash of this node with the node in its left. This algorithm will be the same in each level, not only in level 0.

Case 1: The new node is a left node

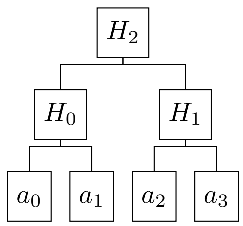
It will not be hashed, it's value will be sent to the next level.

If we add a_4.

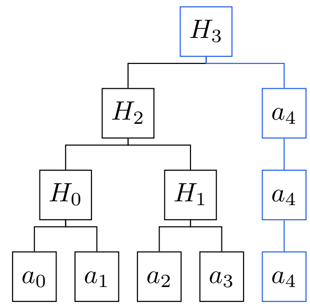
$T = (V, E)$

$V = \{a_0, a_1, a_2, a_3, H_0, H_1, H_2\}$

$E = \{(a_0, H_0), (a_1, H_0), (a_2, H_1), (a_3, H_1), (H_0, H_2), (H_1, H_2)\}$



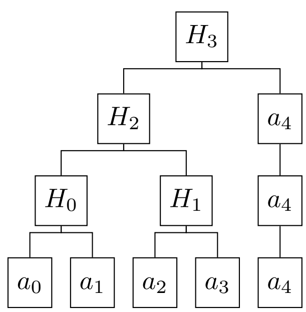
Before inserting a_4



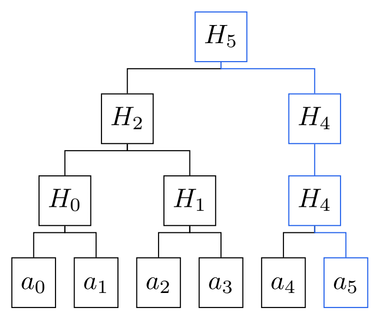
After inserting a_4

Case 2: The new node is a right node

If we add a_5.



Before inserting a_5



After inserting a_5

4.2.1 Pseudocode

Algorithm 1 LeanIMT Insert algorithm

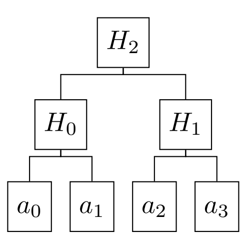
```
procedure Insert(leaf)
    require leaf is defined
    if depth < newDepth then    ▷ newDepth is the new depth of the tree after inserting the new node
        add a new empty array to nodes    ▷ Add a new tree level
    end if
    node ← leaf
    index ← size    ▷ The index of the new leaf equals the number of leaves in the tree.
    for level from 0 to depth - 1 do
        nodes[level][index] ← node
        if index is odd then    ▷ It's a right node
            sibling ← nodes[level][index - 1]
            node ← hash(sibling, node)
        end if
        index ← ⌊index/2⌋    ▷ Divides a number by 2 and discards the remainder.
    end for
    nodes[depth] ← [node]    ▷ Store the new root at the top level
end procedure
```

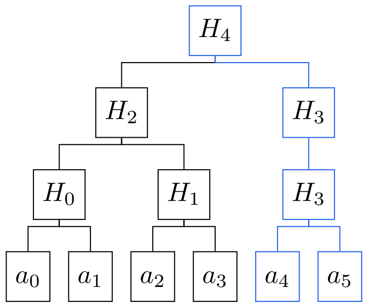
4.3 Batch Insertion

Performing the insertion in bulk rather than individually using a loop can lead to significant performance improvements. This optimization stems from the reduced number of hashing operations required. By inserting many elements at once, the algorithm can minimize redundant computations and manage memory more efficiently, resulting in faster execution and better overall performance.

Insert a_4 and a_5.

Before inserting a_4 and a_5





After inserting a_4 and a_5

TODO: Continue adding the pseudocode to batch insertion

4.3.1 Pseudocode

Algorithm 2 LeanIMT InsertMany algorithm

```
procedure InsertMany(leaves: List of nodes)
    require leaves is defined and it is an array
    if leaves lenght = 0 then
        THROW ERROR "There are no leaves to add"
    end if
    startIndex ← ⌊index/2⌋        ▷ Divides a number by 2 and discards the remainder.
    if depth < newdepth then      ▷ If the tree depth after inserting a node is greater, add a new tree level
        add a new empty array to nodes
    end if
    node ← leaf
    index ← size     ▷ The index of the new leaf equals the number of leaves in the tree.
    for level from 0 to depth - 1 do
        nodes[level][index] ← node
        if index is odd then                         ▷ It's a right node
            sibling ← nodes[level][index - 1]
            node ← hash(sibling, node)
        end if
        index ← ⌊index/2⌋        ▷ Divides a number by 2 and discards the remainder.
    end for
    nodes[depth] ← [node]        ▷ Store the new root at the top level
end procedure
```

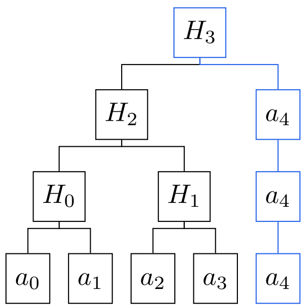
4.4 Update

There are two cases:

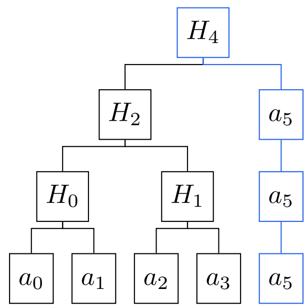
1. When there is no right sibling.
2. When there is right sibling.

Case 1: There is no right sibling

Update a_4 to a_5



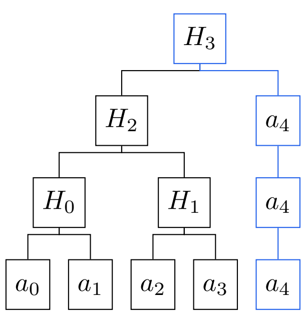
Before updating a_4



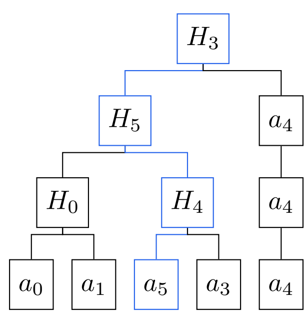
After updating a_4

Case 1: There is right sibling

Update a_2 to a_5



Before updating a_2 for a_5

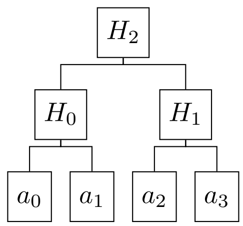


After updating a_2 for a_5

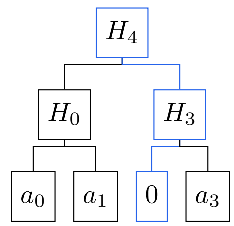
4.5 Remove

The *remove* function is the same as the *update* function but the value used to update is 0.

Remove a_2



Before removing a_2



After removing a_2

4.6 Generate Merkle Proof

4.7 Verify Merkle Proof

5 Implementations

5.1 TypeScript

5.2 Solidity

6 Benchmarks

7 Conslusions

This document is based on the work of [1].

References

[1] Barry Whitehat Kobi Gurkan Koh Wei Jie. “Semaphore: Zero-Knowledge Signaling on Ethereum”. In: (2020). URL: https://semaphore.pse.dev/whitepaper-v1.pdf.