LeanIMT: An optimized Incremental Merkle Tree

Privacy & Scaling Explorations

August 1, 2024

1 Abstract

Created: August 1, 2024

Updated: August 1, 2024

Contents

2 Introduction

2.1 Motivation

2.2 Binary Tree

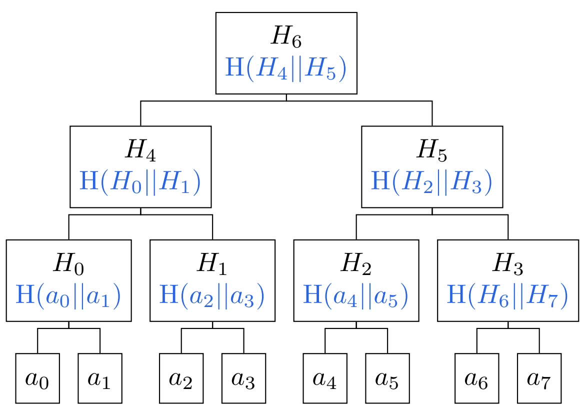
A Binary Tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child. [2]

3 Merkle Tree

A Merkle Tree is a tree (usually a binary tree) in which every leaf is a hash and every node that is not a leaf is the hash of its child nodes. [3]

3.1 Incremental Merkle Tree

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



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.

The tree is always built from the leaves to the root.

The tree will always be balanced by construction.

In a LeanIMT a node is either a leaf or a parent.

Example of a LeanIMT

T - Tree

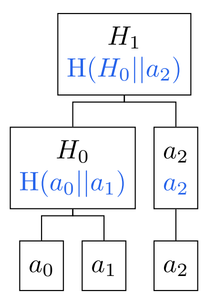
V - Vertices (Nodes)

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

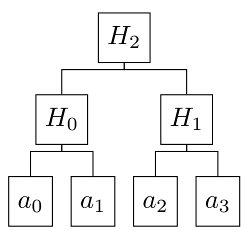
It will not be hashed, it's value will be sent to the next level.

If we add a_4.

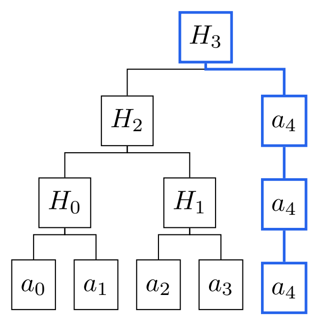
$T = (V, E)$

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

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



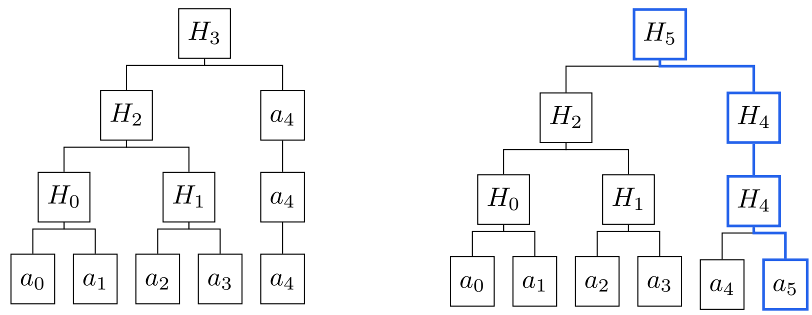
Before inserting a_4



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($leaf$)
  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
  $node \leftarrow leaf$
  $index \leftarrow size$ ▷ 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 ← $\lfloor index/2 \rfloor$ ▷ Divides the index by 2 and discards the remainder.
  end for
  nodes[depth] ← [node] ▷ Store the new root at the top level
end procedure

4.2.2 Correctness

Prove that after inserting a new node to the LeanIMT, the tree keeps all the properties.

The Mathematical Induction method will be used to prove the correctness of this algorithm.

n: Number of leaves in the tree.

Base Case

$n = 0$

If the tree is empty, the Insert function returns a new node with the value of the leaf. This satisfies the LeanIMT poperties because the node does not have children. (Trivial case)

Inductive Hypothesis

Assume that with n nodes, the tree is a correct LeanIMT.

Inductive Step

$n \Rightarrow n + 1$

Assume that with n nodes, the tree is a correct LeanIMT and prove that with $n + 1$ nodes, the tree is still a correct LeanIMT.

The new node is always added at the end of the list of nodes.

To insert a new node, there are two possible cases:

1. When the new node is a left node.
2. When the new node is a right node.

Case 1: The new node is a left node

If the new node is a left node, it means that there were an even number of nodes.

Then, sice it's a left node, the parents has only one child and the parent has the same value as the child which is is the new node. Then all the ancestors will be constructed following the two properties of the LeanIMT.

Since the tree was a correct LeanIMT with n nodes and inserting a new node that is a left node follows the properties of the LeanIMT $\Rightarrow$ the entire tree is still a correct LeanIMT.

Case 2: The new node is a right node

If the new node is a right node, it means that there were an odd number of nodes.

Since it is a right node, the value of the parent will be the hash of the left child with the right child which is the new node. Then all the ancestors will be constructed following the two properties of the LeanIMT.

Since the tree was a correct LeanIMT with n nodes and inserting a new node that is a right node follows the properties of the LeanIMT $\Rightarrow$ the entire tree is still a correct LeanIMT.

Then for the two cases 1 and 2 the Insert algorithm follows the LeanIMT properties $\Rightarrow$ the Insert algorithm is correct.

Conclusion

Then by mathematical induction, the *insert* function works correctly.

4.2.3 Time complexity

n: Number of leaves in the tree.

d: Tree depth.

Every time a new node is added, it is necessary to update or add the ancestors up to the root of the tree.

Number of operations when adding a leaf: $\lceil d \rceil$

$$
\begin{aligned}
\lceil d \rceil &\leq d + 1 \\
&\leq O(\log n) + 1 \\
&\Rightarrow \boxed{O(\log n)}
\end{aligned}
$$

$$
\begin{aligned}
d = \lceil \log(n) \rceil &\leq \log(n) + 1 \\
&\Rightarrow O(\log n)
\end{aligned}
$$

The time complexity of the *Insert* function is $\boldsymbol{O(\log n)}$.

4.3 Batch Insertion

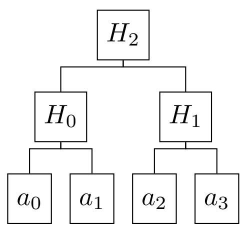
Performing the insertion in bulk rather than individually using a loop can lead to significant performance improvements because the number of hashing operations is reduced.

When inserting n members, all levels will be updated n times if the batch insertion function is not being used.

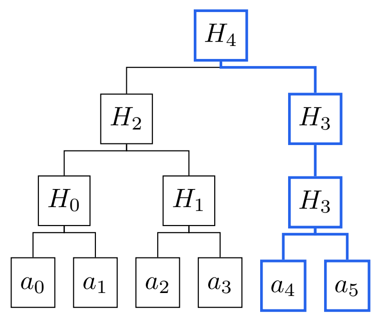
The core idea behind the batch insertion algorithm is to update each level only once even if there are many members to be inserted.

The algorithm will go through the nodes that are necessary to update the next level of the tree. The other nodes in the tree will not change.

Insert a_4 and a_5.



Before inserting a_4 *and* a_5



After inserting a_4 *and* a_5

4.3.1 Pseudocode

Algorithm 2 LeanIMT InsertMany algorithm

```
procedure InsertMany(leaves: List of nodes)
    startIndex ← ⌊size/2⌋    ▷ Divides the size of the tree by 2 and discards the remainder.
    Add leaves to the tree leaves
    for level from 0 to depth - 1 do
        numberOfNodes ← ⌈nodes[level].length/2⌉    ▷ Calculate the number of nodes of the next level. numberOfNodes will be the smallest integrer which is greater than or equal to the result of dividing the number of nodes of the level by 2.
        for index from startIndex to numberOfNodes - 1 do
            rightNode ← nodes[level][index * 2 + 1]    ▷ Get the right node if exists.
            leftNode ← nodes[level][index * 2]    ▷ Get the left node if exists.
            if rightNode exists then
                parentNode ← hash(leftNode, rightNode)
            else
                parentNode ← leftNode
            end if
            nodes[level + 1][index] ← parentNode    ▷ Add the parent node to the tree.
        end for
        startIndex ← ⌊startIndex/2⌋    ▷ Divide startIndex by 2 and discards the remainder.
    end for
end procedure
```

4.3.2 Correctness

To prove the correctness of the $insertMany$ algorithm, the correctness of the $insert$ function will be used.

The $insertMany$ function inserts multiple new leaf nodes into the LeanIMT and updates the tree structure accordingly. The correctness of the algorithm requires verifying that the tree maintains its structure and all nodes correctly reflect the new leaves.

The Mathematical Induction method will be used to prove the correctness of this algorithm.

n: Number of leaves in the tree.

k: Number of leaves to insert into the tree.

Base Case

$k = 1$

The base case is if 1 leaf is inserted. This is equivalent to the *insert* function. If the tree is correct after inserting 1 leaf, then the *insertMany* function is correct for inserting a single leaf. The tree will have $n + 1$ leaves and will be a correct LeanIMT.

Inductive Hypothesis

Assume that the *insertMany* function works correctly for inserting k leaves, because the tree structure is maintained and all nodes are correctly updated up to the root. Then the LeanIMT is correct with $n + k$ leaves.

Inductive Step

$k \Rightarrow k + 1$

Prove that if the *insertMany* function works for inserting k leaves, it also works for inserting $k + 1$ leaves. Then the LeanIMT will be correct with $n + k + 1$ leaves.

Since the tree is initially correct with n leaves and inserting k new leaves using the *insertMany* functions results in a correct LeanIMT with $n + k$ leaves by inductive hypothesis, then inserting 1 additional leaf using the *insert* function results in a correct LeanIMT with $n + k + 1$ leaves because it was proven that the *insert* function is correct.

Then by inductive hypothesis and the correctness of the *insert* function, the entire tree remains correct after inserting $k + 1$ leaves.

Conclusion

By mathematical induction, the *insertMany* function works correctly for inserting any number of leaves into the tree.

4.3.3 Time complexity

n: Number of leaves in the tree.

d: Tree depth.

m: Number of leaves to insert.

Number of operations when inserting elements in batch:

$\lceil m \rceil + \lceil \frac{m}{2} \rceil + \lceil \frac{m}{4} \rceil + ... + \lceil \frac{m}{2^d} \rceil$

That is the same as $\sum_{k=0}^{d} \lceil \frac{m}{2^k} \rceil$

$\lceil m \rceil \leq m + 1$ then $\lceil \frac{m}{2^k} \rceil \leq \frac{m}{2^k} + 1$

$$\begin{aligned}
\sum_{k=0}^{d} \lceil \tfrac{m}{2^k} \rceil &\leq \sum_{k=0}^{d} (\tfrac{m}{2^k} + 1) \\
&\leq \sum_{k=0}^{d} \tfrac{m}{2^k} + \sum_{k=0}^{d} 1 \\
&\leq 2m + O(\log(n+m)) \\
&\leq O(m) + O(\log(n+m)) \\
&\Rightarrow \boxed{O(m)}
\end{aligned}$$

$\sum_{k=0}^{d} \frac{m}{2^k} = m \sum_{k=0}^{d} \frac{1}{2^k} \approx m * 2 \Rightarrow 2m$

$\sum_{k=0}^{d} \frac{1}{2^k}$ (Geometric series)

$|r| < 1$; $r = \frac{1}{2}$

$$\begin{aligned}
\sum_{k=0}^{\infty} a * r^n = \tfrac{a}{1-r} &= \tfrac{1}{1-\frac{1}{2}} \\
&= \tfrac{1}{\frac{1}{2}} \\
&= 2
\end{aligned}$$

$$\begin{aligned}
\sum_{k=0}^{d} 1 = \tfrac{a}{1-r} &= d + 1 \\
&= O(\log(n+m)) + 1 \\
&\Rightarrow O(\log(n+m))
\end{aligned}$$

$$\begin{aligned}
d = \lceil \log(n+m) \rceil &\leq \log(n+m) + 1 \\
&\Rightarrow O(\log(n+m))
\end{aligned}$$

Then the time complexity of the *InsertMany* function is $\boldsymbol{O(m)}$.

Loop Insertion vs Batch Insertion

The time complexity of the Insertion function using a loop is $O(\log(n+m))$.

Going to the root to update or add nodes requires $\log(n+m)$ number of operations.

If we go to the root to update or add nodes m times (one time per leaf to add) then we will have:

$m * \log(n+m) \Rightarrow \boxed{O(m\log(n+m))}$

$\Rightarrow$ Time complexity Loop Insertion is superlinear: $\boldsymbol{O(m\log(n+m))}$

$\Rightarrow$ Time complexity Batch Insertion (*InsertMany* function) is linear: $\boldsymbol{O(m)}$ (linear)

$\Rightarrow$ In terms of time complexity, it is more efficient to use the *InsertMany* function than the *Insert* function in a loop.

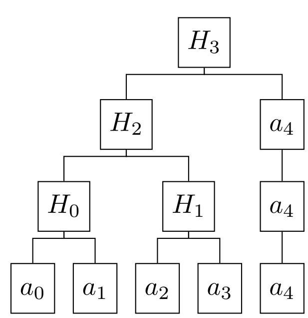
4.4 Update

There are two cases:

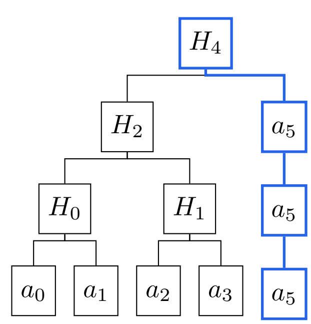
1. When there is no right sibling.
2. When there is right sibling.

Case 1: There is no right sibling

Update a_4 to a_5



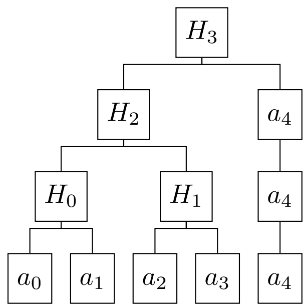
Before updating a_4 to a_5



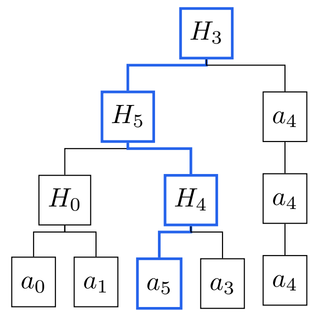
After updating a_4 to a_5

Case 2: There is right sibling

Update a_2 to a_5



Before updating a_2 to a_5



After updating a_2 to a_5

4.4.1 Pseudocode

Algorithm 3 LeanIMT Update algorithm

```
procedure UPDATE(index, newLeaf)
    node ← newLeaf
    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)
        else                                              ▷ It's a left node
            sibling ← nodes[level][index + 1]
            if sibling exists then        ▷ It's a left node with a right sibling
                node ← hash(node, sibling)
            end if
        end if
        index ← ⌊index/2⌋        ▷ Divides the index by 2 and discards the remainder.
    end for
    nodes[depth] ← [node]               ▷ Store the new root at the top level
end procedure
```

4.4.2 Correctness

Prove that after updating a node in the LeanIMT, the tree keeps all the properties.

The Mathematical Induction method will be used to prove the correctness of this algorithm.

The induction will be done using the depth of the tree. The correctness of the update function will be proven for all tree depths.

d: Depth of the tree.

Base Case

$d = 0$

If the depth is 0, it can have 0 or 1 node. If it has 0 leaves it is not necessary to use this function because there are no leaves. If it has 1 node, it is the root of the tree and if it is updated, the root is also updated because they are the same node.

Inductive Hypothesis

Assume that for a LeanIMT with depth d, the *update* function works correctly because after updating a leaf everything is updated correctly up to the root.

Inductive Step

$d \Rightarrow d + 1$

If for a LeanIMT with depth d, the *update* function works correctly because after updating a leaf everything is updated correctly up to the root, then prove that for depth $d + 1$ the update function also works correctly.

There are two cases when updating a leaf:

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

Case 1: There is no right sibling

If the leaf that will be updated does not have a right sibling, then the parent node will have the same value as the leaf node.

Case 2: There is right sibling

If the leaf node has a right sibling, the parent value will be the hash of the leaf node with its right sibling.

After those cases, the tree that should be updated has depth d. Using the inductive hypothesis, the subtree of depth d will correctly update all the ancestors up to the root.

Conclusion

Then by mathematical induction, the update function works correctly for any depth of the tree.

4.4.3 Time complexity

n: Number of leaves in the tree.

d: Tree depth.

Every time a leaf is updated, it is necessary to update all the ancestors up to the root of the tree.

Number of operations when updating a leaf: $\lceil d \rceil$

This proof is the same as the proof of the time complexity of the Insert function.

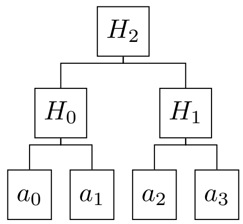
The time complexity of the *Update* function is $\boldsymbol{O(\log n)}$.

4.5 Remove

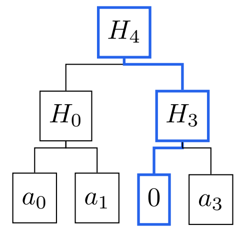
The *remove* function is the same as the *update* function but the value used to update is 0.

You can use a value other than 0, the idea is to use a value that is not a possible value for a correct member in the list.

Remove a_2



Before removing a_2



After removing a_2

4.5.1 Pseudocode

Algorithm 4 LeanIMT Remove algorithm

procedure REMOVE($index$)
  $update(index, 0)$
end procedure

4.5.2 Correctness

The proof of the correctness of this algorithm is the same as the Update function.

4.5.3 Time complexity

The proof of the time complexity of this algorithm is the same as the Update function.

4.6 Generate Merkle Proof

There are two cases:

1. When the node does not have a sibling.
2. When the node has a sibling.

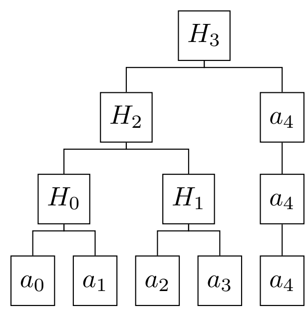
d: Depth of the tree

Case 1 only happens when the node is the last node in the level and also a left node.

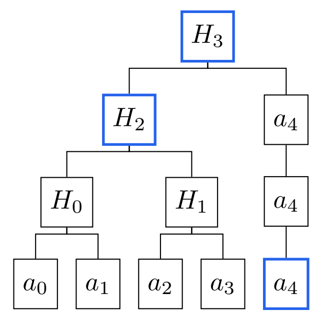
We will always see one of these two cases. When you want to generate the proof for the case 1, there will always be one node in the siblings list, for the case 2 there will always be $\lceil d \rceil$ number of nodes.

Case 1: When the node does not have a sibling

If we want to generate a proof for the node a_4.



LeanIMT to generate a proof for a_4



Nodes used to generate a proof for a_4

path: [1]

Merkle Proof: {

root: H_3

leaf: a_4

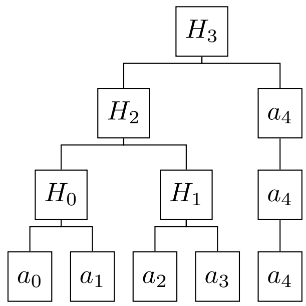
index: 1

siblings: [H_2]

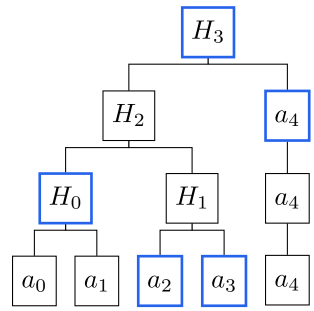
}

Case 2: When the node has a sibling

If we want to generate a proof for the node a_3.



LeanIMT to generate a proof for a_3



Nodes used to generate a proof for a_3

path: [1, 1, 0]

Merkle Proof: {

root: H_3

leaf: a_3

index: 3

siblings: [a_2, H_0, a_2]

}

4.6.1 Pseudocode

Algorithm 5 LeanIMT generateProof algorithm

```
procedure generateProof(index)
    siblings ← empty list ▷ List to store the nodes necessary to rebuild the root.
    path ← empty list ▷ List of 0s or 1s to help rebuild the root. 0 if the current node is a left node and the sibling is a right node and 1 otherwise.
    for level from 0 to depth - 1 do
        isRightNode ← index is odd
        if isRightNode is true then ▷ It's a right node
            siblingIndex ← index - 1
        else ▷ It's a left node
            siblingIndex ← index + 1
        end if
        sibling ← nodes[level][siblingIndex]
        if sibling exists then
            add isRightNode to path
            add sibling to siblings
        end if
        index ← ⌊index/2⌋ ▷ Divides the index by 2 and discards the remainder.
    end for
    leaf ← leaves[index]
    index ← reverse path and use the list as a binary number and get the decimal representation
    siblings ← leaves[index]
    proof ← {root, leaf, index, siblings}
    return proof
end procedure
```

4.6.2 Correctness

Prove that the $generateProof$ function works correctly for all tree depths.

The Mathematical Induction method will be used to prove the correctness of this algorithm.

The induction will be done using the depth of the tree. The correctness of the $generateProof$ function will be proven for all tree depths.

d: Depth of the tree.

Base Case

$d = 0$

If the depth is 0, it can have 0 or 1 node. If it has 0 leaves it is not necessary to use this function because there are no leaves. If it has 1 leaf, the merkle proof will have the root and the leaf with the same values because they are the same nodes. The siblings will be empty and the path (index) too.

Inductive Hypothesis

Assume that for a LeanIMT with depth d, the $generateProof$ function works correctly because it returns the correct values necessary to rebuild the root.

Inductive Step

$d \Rightarrow d + 1$

If for a LeanIMT with depth d, the $generateProof$ function works correctly, prove that for a tree with depth $d + 1$, the $generatProof$ function also works correctly.

There are two cases when generating a proof:

1. When the node does not have a sibling.
2. When the node has a sibling.

Case 1: The node does not have a sibling

If the node does not have a sibling, nothing is added to the proof.

Case 2: The node has a sibling

If the node has a sibling, the path and siblings arrays are updated accordingly. If the node is a right node, 1 will be added to the path and the left sibling will be added to siblings. If the node is a left node, 0 will be added to the path and the right sibling will be added to siblings.

After those cases, the tree that should generate the proof has depth d. Using the inductive hypothesis, the subtree of depth d will correctly return the values in the proof.

Conclusions

By mathematical induction, the $generateProof$ function works correctly for any depth of the tree.

4.6.3 Time complexity

n: Number of leaves in the tree.

d: Tree depth.

To generate a Merkle Proof it is necessary to visit all the ancestors of the leaf up to the root of the tree.

Number of operations to generate a Merkle Proof: $\lceil d \rceil$

This proof is the same as the proof of the time complexity of the Insert function.

The time complexity of the *generateProof* function is $\boldsymbol{O(\log n)}$.

4.7 Verify Merkle Proof

The $verifyProof$ function will verify if a leaf is part of a tree having a Merkle Proof.

The algorithm will go through the sibling nodes using the path and calculate the parent in the next level of the tree. Then it will check if the calculated root matches the one that is part of the proof. If the calculates root matches the one that is part of the root, the algorithm will return true, otherwise it will return false.

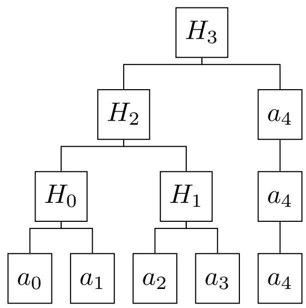
There are two example of cases:

1. When the node does not have a sibling.
2. When the node has a sibling.

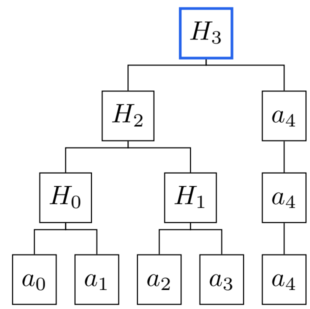
Case 1: When the node does not have a sibling

If we want to verify a proof for the node a_4.

All the parent hashes will be calculated. Something to notice here is that only one hash operation will be performed.



LeanIMT to verify a proof for a_4



Nodes rebuilt to verify a proof for a_4

path: [1]

Merkle Proof: {

root: H_3

leaf: a_4

index: 1

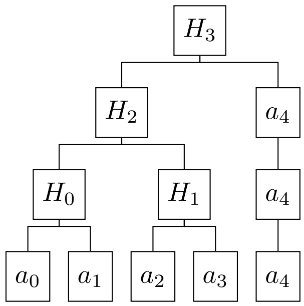
siblings: [H_2]

}

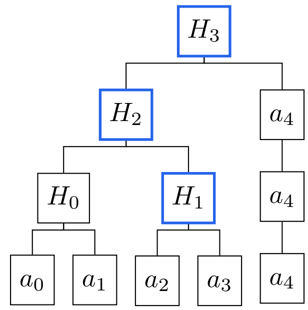
Case 2: When the node has a sibling

If we want to verify a proof for the node a_3.

All the parent hashes will be calculated.



LeanIMT to verify a proof for a_3



Nodes rebuilt to verify a proof for a_3

path: [1, 1, 0]

Merkle Proof: {

root: H_3

leaf: a_3

index: 3

siblings: [a_2, H_0, a_2]

}

4.7.1 Pseudocode

Algorithm 6 LeanIMT verifyProof algorithm

```
procedure verifyProof(index)
    { root, leaf, siblings, index } ← proof            ▷ Deconstruct the proof
    node ← leaf
    for i from 0 to siblings.length - 1 do
        isOdd ← devide index by 2 i times and check if the result is odd
        if isOdd is true then                           ▷ node is a right child
            node ← hash(siblings[i], node)
        else                                            ▷ It's a left node
            node ← hash(node, siblings[i])
        end if
    end for
    if root is equal node then
        return true
    else
        return false
    end if
end procedure
```

4.7.2 Correctness

Prove that the $verifyProof$ function works correctly for all tree depths.

The Mathematical Induction method will be used to prove the correctness of this algorithm.

The induction will be done using the depth of the tree. The correctness of the $verifyProof$ function will be proven for all tree depths.

d: Depth of the tree.

Base Case

$d = 0$

If the depth is 0, it can have 0 or 1 node. If it has 0 leaves it is not necessary to use this function because there are no leaves. If it has 1 leaf, the merkle root will be equal to the node so it will verify the proof successfully.

Inductive Hypothesis

Assume that for a LeanIMT with depth d, the $verifyProof$ function works correctly because it knows how to calculate all the hashes up to the root and then check if the calculated hash is equal to the root in the proof.

Inductive Step

$d \Rightarrow d + 1$

If for a LeanIMT with depth d, the $verifyProof$ function works correctly, prove that for a tree with depth $d+1$, the $verifyProof$ function also works correctly.

If the current node (which is the leaf at the beginning of the algorithm) is a right node, the parent value will be the hash of the left sibling with the current node. If the current node is a left node, the parent value will be the hash of the current node with the right sibling. Then, the current node will be the parent node.

After running that once, the tree that should verify the proof has depth d. Using the inductive hypothesis, the subtree of depth d will correctly return the values in the proof.

Conclusion

Then the $verifyProof$ function works correctly for any depth of the tree.

4.7.3 Time complexity

n: Number of leaves in the tree.

d: Tree depth.

To verify a Merkle Proof it is necessary to visit (rebuild) all the ancestors of the leaf up to the root of the tree.

Number of operations to verify a Merkle Proof: $\lceil d \rceil$

This proof is the same as the proof of the time complexity of the Insert function.

The time complexity of the *verifyProof* function is $\boldsymbol{O(\log n)}$.

5 Implementations

The TypeScript/JavaScript and Solidity implementations follow the same idea and are compatible but are different.

The TypeScript/JavaScript implementation focuses on performance whereas the Solidity one focuses on saving gas costs.

The TypeScript/JavaScript and Solidity code of the LeanIMT was audited as part of the Semaphore v4 audit [4].

5.1 TypeScript/JavaScript

TypeScript/JavaScript LeanIMT code: https://github.com/privacy-scaling-explorations/zk-kit/tree/main/packages/lean-imt

5.2 Solidity

Solidity LeanIMT code:
https://github.com/privacy-scaling-explorations/zk-kit.solidity/tree/main/packages/lean-imt

6 Benchmarks

All the benchmarks were run in an environment with these properties:

System Specifications

Computer: MacBook Pro

Chip: Apple M2 Pro

Memory (RAM): 16 GB

Operating System: macOS Sonoma version 14.5

Software environment

Node.js version: 20.5.1

Browser: Google Chrome Version 127.0.6533.73 (Official Build) (arm64)

6.1 Running the benchmarks

TypeScript/JavaScript

GitHub repository to run Node.js and browser benchmarks:
https://github.com/vplasencia/imt-benchmarks.

Solidity

GitHub repository to run Solidity benchmarks:
https://github.com/privacy-scaling-explorations/zk-kit.solidity

6.2 TypeScript/JavaScript

Note: The IMT has a static depth. To run the benchmarks, the minimum depth necessary to perform the operation was used unless a specific tree depth was specified.

For example:

- If I the IMT has 4 members and I want to add 1 new member the tree depth used will be 3.
- If I the IMT has 5 members and I want to add 1 new member the tree depth used will be 3.

6.2.1 Node.js

Table 1: All Functions (100 iterations)

Function	ops/sec	Average Time (ms)	Relative to IMT
IMT - Insert	1287	0.77687	
LeanIMT - Insert	2358	0.42391	1.83 x faster
IMT - InsertMany	12	77.98467	
LeanIMT - InsertMany	144	6.94025	11.24 x faster
IMT - Update	1283	0.77933	
LeanIMT - Update	1223	0.81708	1.05 x slower
IMT - Remove	1306	0.76554	
LeanIMT - Remove	1301	0.76838	1.00 x slower
IMT - GenerateProof	300868	0.00332	
LeanIMT - GenerateProof	321586	0.00311	1.07 x faster
IMT - VerifyProof	1331	0.75121	
LeanIMT - VerifyProof	1336	0.74810	1.00 x faster

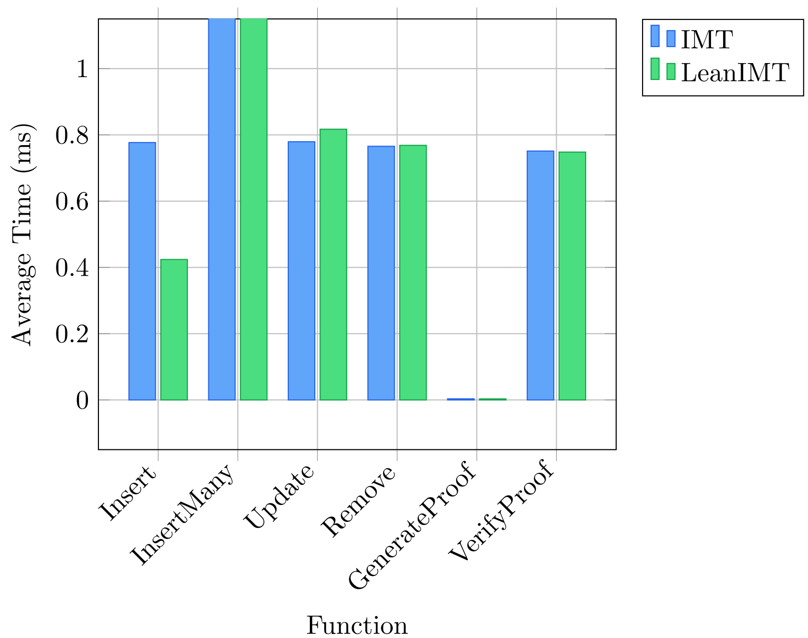


Figure 1: Functions IMT vs LeanIMT (100 iterations)

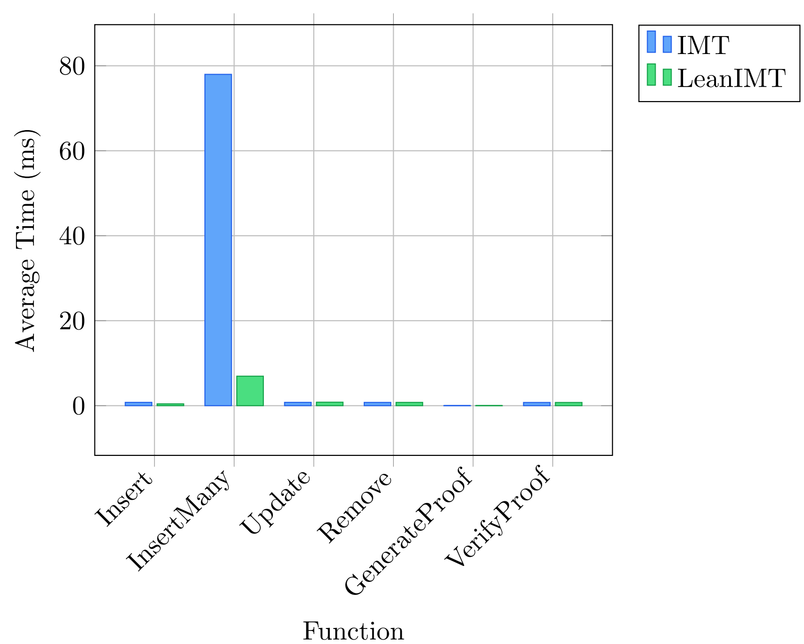


Figure 2: Functions IMT vs LeanIMT (100 iterations)

6.2.2 Browser

Table 2: All Functions (100 iterations)

Function	ops/sec	Average Time (ms)	Relative to IMT
IMT - Insert	1107	0.90300	
LeanIMT - Insert	2590	0.38600	2.34 x faster
IMT - InsertMany	14	68.53200	
LeanIMT - InsertMany	158	6.30200	10.87 x faster
IMT - Update	1455	0.68700	
LeanIMT - Update	1470	0.68000	1.01 x faster
IMT - Remove	1438	0.69500	
LeanIMT - Remove	1472	0.67900	1.02 x faster
IMT - GenerateProof	1000000	0.00100	
LeanIMT - GenerateProof	1000000	0.00100	1.00 x slower
IMT - VerifyProof	1472	0.67900	
LeanIMT - VerifyProof	1508	0.66300	1.02 x faster

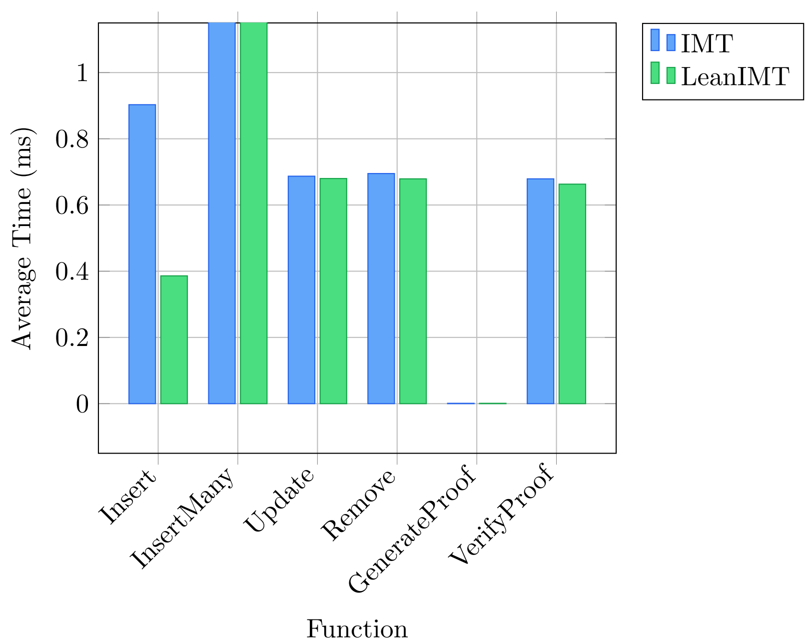


Figure 3: Functions IMT vs LeanIMT (100 iterations)

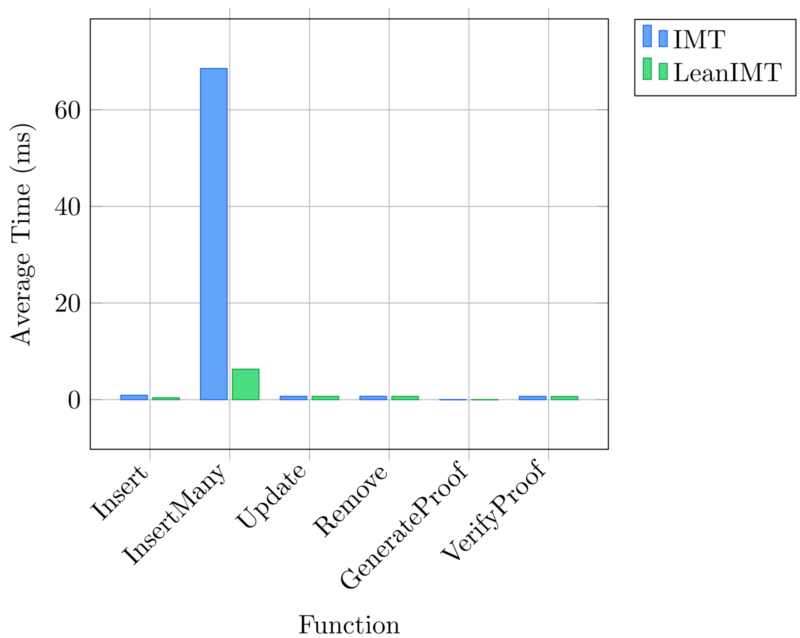


Figure 4: Functions IMT vs LeanIMT (100 iterations)

6.2.3 LeanIMT: Node.js vs Browser

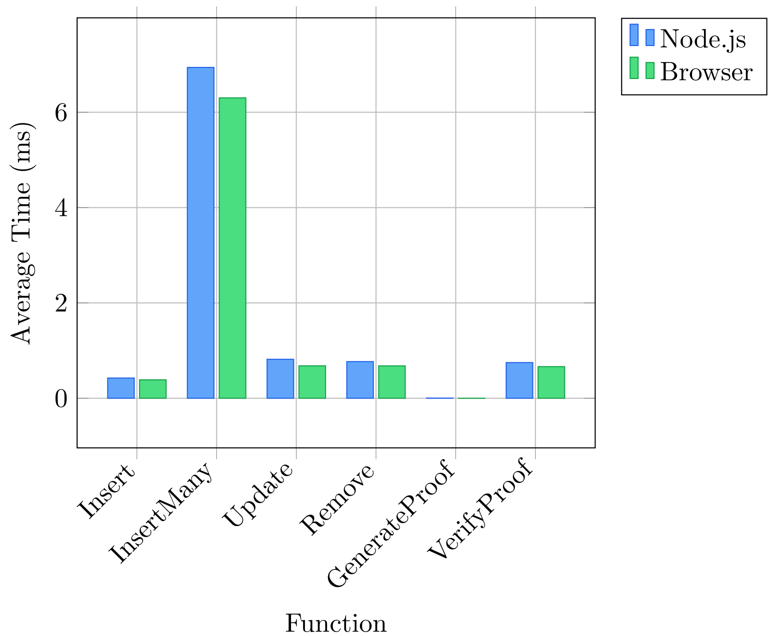


Figure 5: LeanIMT Node.js vs Browser (100 iterations)

6.2.4 Insert Function: IMT vs LeanIMT

Table 3: Insert Function (1000 iterations)

Function	ops/sec	Average Time (ms)	Relative to IMT
IMT	814	1.22803	
LeanIMT	1453	0.68790	1.79 x faster

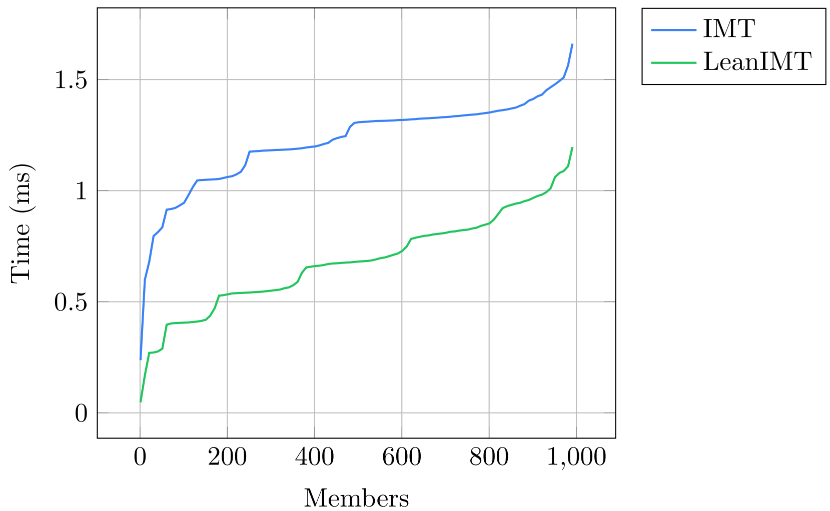


Figure 6: Insert function IMT vs LeanIMT (1000 iterations)

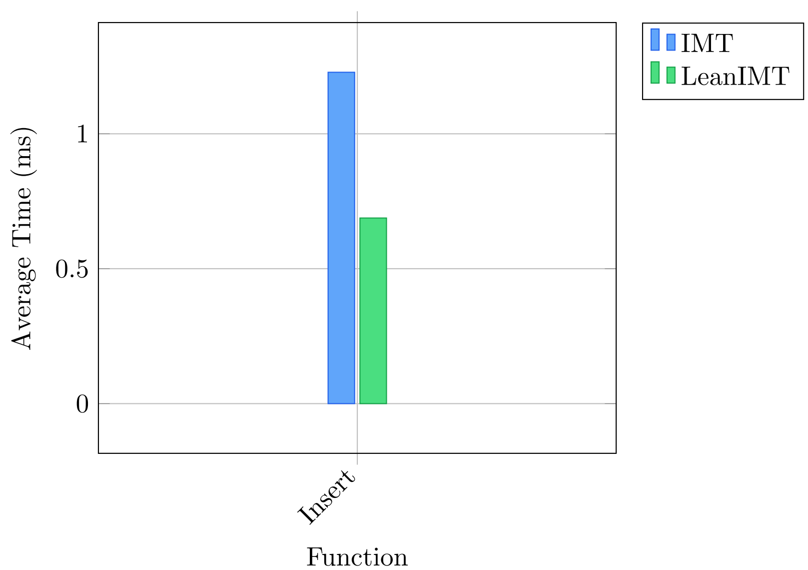


Figure 7: Insert function IMT vs LeanIMT (1000 iterations)

6.2.5 LeanIMT: Insert Loop vs Batch Insertion

Table 4: Insert Function (1000 iterations)

Function	ops/sec	Average Time (ms)	Relative to Insert
Insert in Loop	47	20.97820	
InsertMany	136	7.31698	2.87 x faster

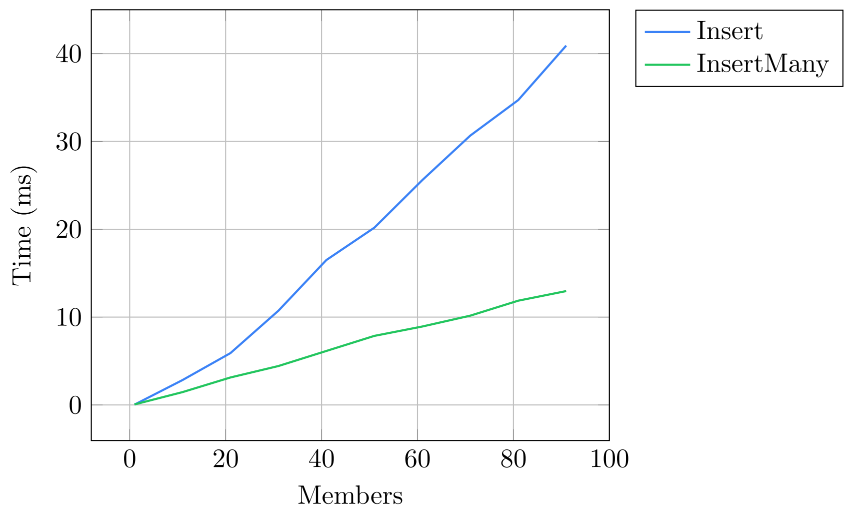


Figure 8: Batch Insertion LeanIMT

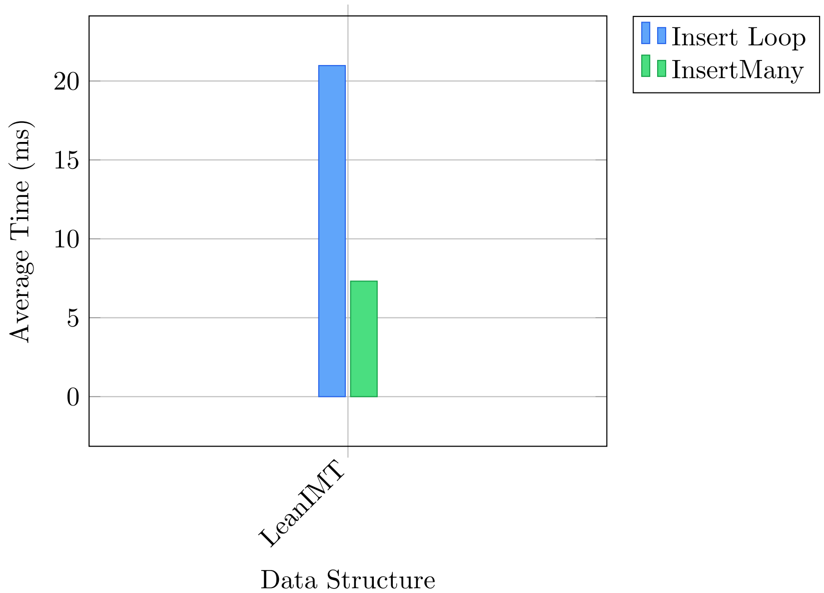


Figure 9: Batch Insertion LeanIMT (1000 iterations)

6.3 Solidity

Solidity and Network Configuration

Solidity: 0.8.23	Optim: true	Runs: 200	viaIR: false	Block: 30,000,000 gas

Methods

Contracts / Methods	Min	Max	Avg	# calls	usd (avg)
BinaryIMTTest					
init	105,471	374,307	357,505	16	-
initWithDefaultZeroes	91,272	91,870	91,471	3	-
insert	98,112	2,501,619	560,351	31	-
remove	471,034	473,216	472,710	7	-
update	-	-	474,000	1	-

Deployments				% of limit	
BinaryIMT	1,237,933	1,238,005	1,237,998	4.1 %	-
BinaryIMTTest	378,277	378,337	378,329	1.3 %	-
PoseidonT3	-	-	3,693,362	12.3 %	-

Key

○ Execution gas for this method does not include intrinsic gas overhead

△ Cost was non-zero but below the precision setting for the currency display (see options)

Toolchain: hardhat

Figure 10: IMT Gas Report

Solidity and Network Configuration					
Solidity: 0.8.23	Optim: true	Runs: 200	viaIR: false	Block: 30,000,000 gas	
Methods					
Contracts / Methods	**Min**	**Max**	**Avg**	**# calls**	**usd (avg)**
LeanIMTTest					
insert	93,938	163,708	119,051	47	-
insertMany	95,891	715,164	322,619	7	-
remove	104,558	296,279	233,235	13	-
update	58,909	252,738	197,830	8	-
Deployments				**% of limit**	
LeanIMT	1,018,010	1,018,082	1,018,077	3.4 %	-
LeanIMTTest	455,827	455,911	455,908	1.5 %	-
PoseidonT3	-	-	3,693,362	12.3 %	-
Key					
○ Execution gas for this method does not include intrinsic gas overhead					
△ Cost was non-zero but below the precision setting for the currency display (see options)					
Toolchain: hardhat					

Figure 11: LeanIMT Gas Report

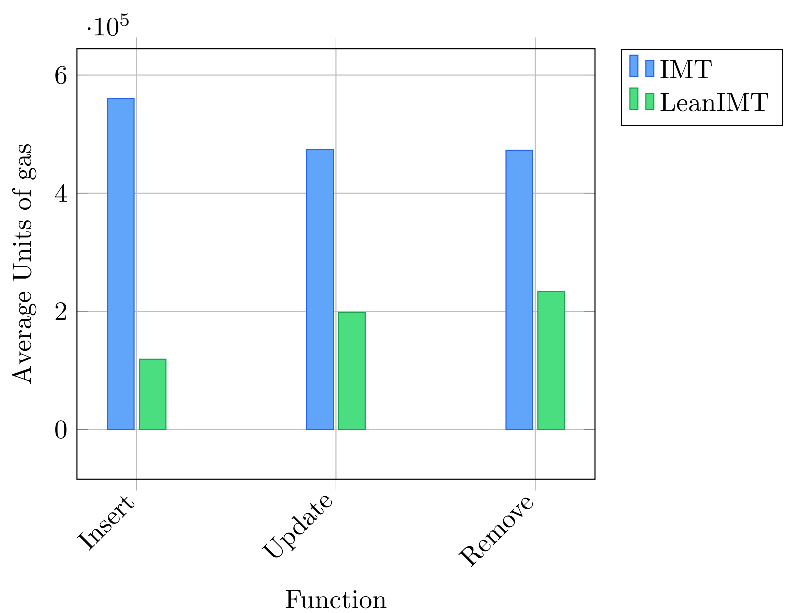


Figure 12: Gas cost of the execution of the Functions IMT vs LeanIMT

7 Conclusions

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.

[2] NIST. "Binary Tree". In: (2017). URL: https://xlinux.nist.gov/dads/HTML/binarytree.html.

[3] NIST. "Merkle Tree". In: (2019). URL: https://xlinux.nist.gov/dads/HTML/MerkleTree.html.

[4] PSE. "Semaphore v4 Audit Report". In: (2024). URL: https://semaphore.pse.dev/Semaphore_4.0.0_Audit.pdf.