LeanIMT: An optimized Incremental Merkle Tree

Privacy Stewards of Ethereum

Cedoor me@cedoor.dev

Vivian Plasencia vivianpc@ethereum.org

September 18, 2024

Abstract

This technical document presents the LeanIMT (Lean Incremental Merkle Tree), a data structure used to represent a group of elements efficiently. The LeanIMT is designed to optimize performance and reduce gas costs, making it suitable for zero-knowledge [18] protocols and applications.

Created: September 18, 2024

Updated: September 18, 2025

Contents

1 Introduction

This technical document aims to present and explain a new data structure called LeanIMT (Lean Incremental Merkle Tree). It covers the definition of Merkle Tree (MT), Incremental Merkle Tree (IMT) and Lean Incremental Merkle Tree (LeanIMT). It also explains the motivation behind creating this data structure and provides detailed algorithm explanations with pseudocode and time complexity analyses. It also includes a section on benchmarks to illustrate performance improvements.

1.1 Motivation

The main motivation for the creation of this data structure was the development of the new version of the Semaphore protocol [5] [16] (version 4). Semaphore version 3 uses an IMT [2] [1], which is, however, rather inefficient and expensive when inserting the first leaves and when the number of leaves exceeds the maximum size supported by the tree.

2 Merkle Tree

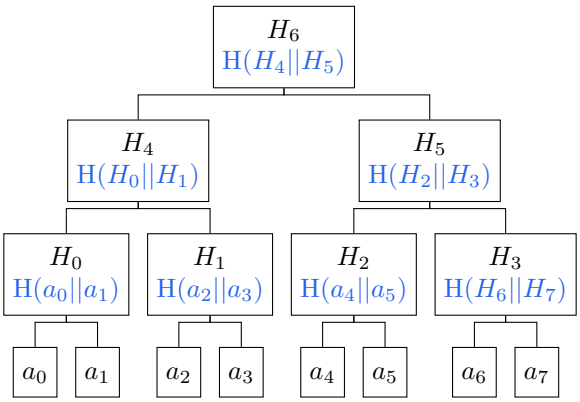
A Merkle Tree (MT) is a tree (usually a binary tree) in which every leaf is a hash and every node that is not a leaf is the hash of its child nodes. [14] [4] [17]

2.1 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. [13]

2.2 Incremental Merkle Tree

An Incremental Merkle Tree (IMT) is a Merkle Tree (MT) designed to be updated efficiently by only appending new entries to the right-hand side. [3]



Example of IMT

3 LeanIMT

3.1 Definition

The **LeanIMT** (Lean Incremental Merkle Tree) is a Binary IMT, inspired by the IMT used in Semaphore v3 [2] [1], as well as Merkle Mountain Range (MMR) [12] [10] [9] [11] and LazyTower [8] [7] [6].

The LeanIMT has two properties:

1. Every node with two children is the hash of its left and right 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.

The tree depth is dynamic and can increase with the insertion of new leaves.

Example of a LeanIMT

T - Tree

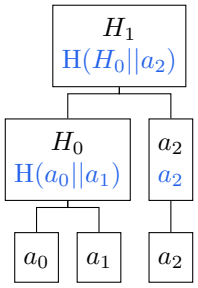
V - Vertices (Nodes)

E - Edges (Lines connecting Nodes)

$T = (V, E)$

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

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



Example of LeanIMT

Note: To calculate a parent hash with two children, always start with the left child followed by the right. The order is never reversed.

3.2 Insertion

Function to insert a new leaf into a LeanIMT.

One of these cases will always be seen in each level when inserting a node:

1. The new node is the left child.
2. The new node is the right child.

Case 1: The new node is a left child

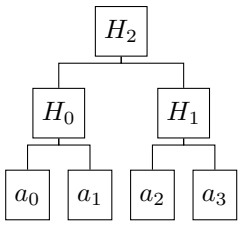
It will not be hashed, its value will be sent to the next level.

Adding a_4.

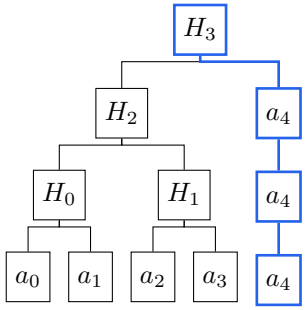
$T = (V, E)$

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

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



Before inserting a_4

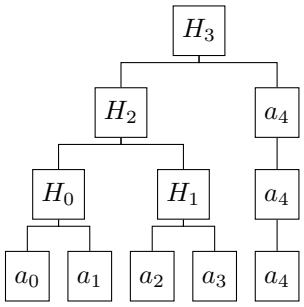


After inserting a_4

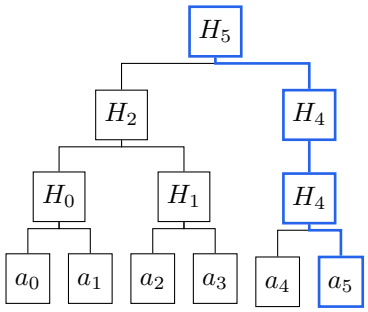
Case 2: The new node is a right child

The parent node will be the hash of the node's sibling with itself.

If we add a_5.



Before inserting a_5



After inserting a_5

3.2.1 Pseudocode

Algorithm 1 LeanIMT Insert algorithm

```
1: procedure INSERT(leaf)
2:     if depth < newDepth then    ▷ newDepth is the new depth of the tree after inserting the new node
3:         add a new empty array to nodes                ▷ Add a new tree level
4:     end if
5:     node ← leaf
6:     index ← size    ▷ The index of the new leaf equals the number of leaves in the tree.
7:     for level from 0 to depth - 1 do
8:         nodes[level][index] ← node
9:         if index is odd then                          ▷ It's a right node
10:            sibling ← nodes[level][index - 1]
11:            node ← hash(sibling, node)
12:        end if
13:        index ← ⌊index/2⌋          ▷ Divides the index by 2 and discards the remainder.
14:    end for
15:    nodes[depth] ← [node]                ▷ Store the new root at the top level
16: end procedure
```

3.2.2 Time complexity

n: Number of leaves in the tree.

d: Tree depth after inserting the new leaf.

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: $d + 1$

$$d + 1 = O(\log n) + 1$$

$$\Rightarrow \boxed{O(\log n)}$$

$$d = \lceil \log(n+1) \rceil \leq \log(n+1) + 1$$

$$\leq O(\log n) + 1$$

$$\Rightarrow O(\log n)$$

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

3.3 Batch Insertion

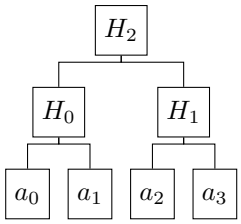
Performing the insertion in bulk rather than individually using a loop can lead to significant performance improvements because the number of hashing operations is reduced.

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

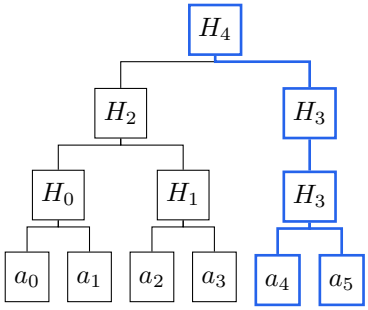
The core idea behind the batch insertion algorithm is to update each level only once even if there are many members to be inserted.

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

3.3.1 Pseudocode

Algorithm 2 LeanIMT InsertMany algorithm

```
1: procedure InsertMany(leaves: List of nodes)
2:     startIndex ← ⌊size/2⌋ ▷ Divides the size of the tree by 2 and discards the remainder. startIndex is the index to start updating values at the next level.
3:     Add leaves to the tree leaves
4:     for level from 0 to depth - 1 do
5:         numberOfNodes ← ⌈nodes[level].length/2⌉ ▷ Calculate the number of nodes of the next level. numberOfNodes will be the smallest integer which is greater than or equal to the result of dividing the number of nodes of the level by 2.
6:         for index from startIndex to numberOfNodes - 1 do
7:             rightNode ← nodes[level][index * 2 + 1] ▷ Get the right node if exists.
8:             leftNode ← nodes[level][index * 2] ▷ Get the left node if exists.
9:             if rightNode exists then
10:                parentNode ← hash(leftNode, rightNode)
11:            else
12:                parentNode ← leftNode
13:            end if
14:            nodes[level + 1][index] ← parentNode ▷ Add the parent node to the tree.
15:        end for
16:        startIndex ← ⌊startIndex/2⌋ ▷ Divide startIndex by 2 and discards the remainder.
17:    end for
18: end procedure
```

3.3.2 Time complexity

n: Number of leaves in the tree.

m: Number of leaves to insert.

d: Tree depth after inserting the m new leaves.

The number of operations required for batch insertion of elements is **approximately**:

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

Note:

This expression represents the exact number of operations when reconstructing the tree from scratch.

The term **approximately** is used because the expression reflects the process of pairing nodes to create the parent nodes, assuming that, at each level, the first node to be paired is always a left child. This scenario typically occurs when the $insertMany$ function is used to add new elements to an empty tree or a tree that already has a power of two leaves.

However, when the first node to be paired at level $k-1$ (assuming that the level of the leaves is 0, then $k > 0$) is a right child, the number of operations required to compute the parent nodes of that level will be $\lceil \frac{m}{2^k} \rceil + 1$. For example, if a tree has 3 leaves and you want to add 2 new ones, updating the parent nodes of the new leaves (level 1) will require 2 operations instead of 1.

When these additional 1s are summed across levels, the total is $< log(n+m)$ which can be included in the initial expression as $+log(n+m)$ without affecting the overall time complexity of the $insertMany$ function.

The expression $m + \lceil \frac{m}{2} \rceil + \lceil \frac{m}{4} \rceil + ... + \lceil \frac{m}{2^d} \rceil$ is the same as $\sum_{k=0}^{d} \lceil \frac{m}{2^k} \rceil$

Note: $m = \lceil m \rceil$

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

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

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

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

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

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

$\sum_{k=0}^{d} 1 = d + 1$

$$= O(\log(n+m)) + 1$$

$$\Rightarrow O(\log(n+m))$$

$$d = \lceil \log(n+m) \rceil \leq \log(n+m) + 1$$

$$\Rightarrow O(\log(n+m))$$

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

Loop Insertion vs Batch Insertion

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

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

If going to the root to update or add nodes m times (one time per leaf to add) then the result will be:

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

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

$\Rightarrow$ Time complexity Batch Insertion (*InsertMany* function):
$\boldsymbol{O(m + \log(n+m))}$

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

3.4 Update

Function to update the value of a leaf of a LeanIMT.

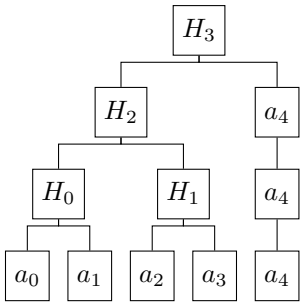
There are two cases:

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

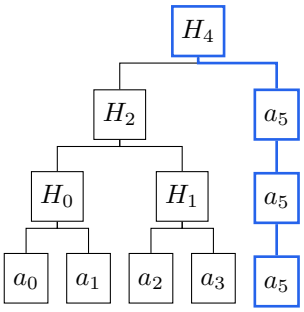
Case 1: The node does not have a sibling

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

Update a_4 to a_5



Before updating a_4 to a_5

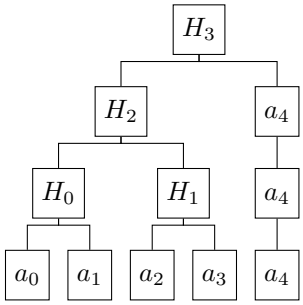


After updating a_4 to a_5

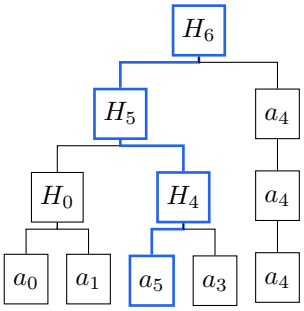
Case 2: The node has a sibling

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

Update a_2 to a_5



Before updating a_2 to a_5



After updating a_2 to a_5

3.4.1 Pseudocode

Algorithm 3 LeanIMT Update algorithm

```
1: procedure UPDATE(index, newLeaf)
2:     node ← newLeaf
3:     for level from 0 to depth - 1 do
4:         nodes[level][index] ← node
5:         if index is odd then                          ▷ It's a right node
6:             sibling ← nodes[level][index - 1]
7:             node ← hash(sibling, node)
8:         else                                          ▷ It's a left node
9:             sibling ← nodes[level][index + 1]
10:            if sibling exists then          ▷ It's a left node with a right sibling
11:                node ← hash(node, sibling)
12:            end if
13:        end if
14:        index ← ⌊index/2⌋          ▷ Divides the index by 2 and discards the
   remainder.
15:    end for
16:    nodes[depth] ← [node]                ▷ Store the new root at the top level
17: end procedure
```

3.4.2 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: $d + 1$

$d + 1 = O(\log n) + 1$

$$\Rightarrow \boxed{O(\log n)}$$

$d = \lceil \log(n) \rceil \leq \log(n) + 1$

$$\leq O(\log n) + 1$$

$$\Rightarrow O(\log n)$$

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

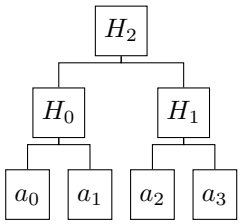
3.5 Remove

Function to remove a leaf from a LeanIMT.

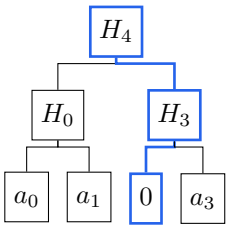
The *remove* function is the same as the *update* function but the value used to update is 0.

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

3.5.1 Pseudocode

Algorithm 4 LeanIMT Remove algorithm

```
1: procedure REMOVE(index)
2:     update(index, 0)
3: end procedure
```

3.5.2 Time complexity

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

3.6 Generate Merkle Proof

Function to generate a Merkle Proof of a leaf in a LeanIMT.

There are two cases:

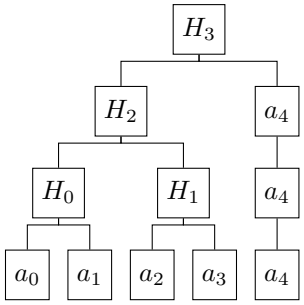
1. When the node does not have a sibling.
2. When the node has a sibling.

Case 1: The node does not have a sibling

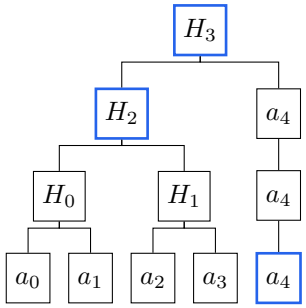
When generating the proof for this case, nothing is added to the proof at that level.

This case only happens when the node is the last node in the level and is also a left node.

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

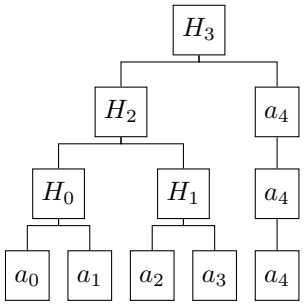
siblings: $[H_2]$

}

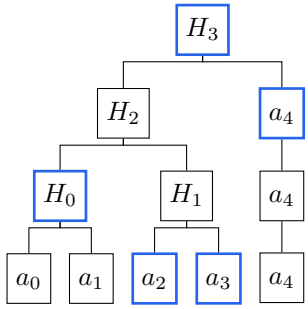
Case 2: The node has a sibling

When generating the proof for this case, if the node is a right node, 1 will be added to the path and the left sibling will be added to siblings and if the node is a left node, 0 will be added to the path and the right sibling will be added to siblings.

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_4]

}

3.6.1 Pseudocode

Algorithm 5 LeanIMT generateProof algorithm

```
1: procedure GENERATEPROOF(index)
2:     siblings ← empty list ▷ List to store the nodes necessary to rebuild the root.
3:     path ← empty list ▷ List of 0s or 1s to help rebuild the root. 0 if the current node is a left node with a sibling and 1 if the current node is a right node.
4:     for level from 0 to depth - 1 do
5:         isRightNode ← index is odd
6:         if isRightNode is true then ▷ It's a right node
7:             siblingIndex ← index - 1
8:         else ▷ It's a left node
9:             siblingIndex ← index + 1
10:        end if
11:        sibling ← nodes[level][siblingIndex]
12:        if sibling exists then
13:            add isRightNode to path
14:            add sibling to siblings
15:        end if
16:        index ← ⌊index/2⌋ ▷ Divides the index by 2 and discards the remainder.
17:    end for
18:    leaf ← leaves[index]
19:    index ← reverse path and use the list as a binary number and get the decimal representation
20:    siblings ← leaves[index]
21:    proof ← {root, leaf, index, siblings}
22:    return proof
23: end procedure
```

Note:

The *index* parameter of the function and the *index* returned in the proof are different variables that share the same name but may not always have same value. The *index* in the parameter is the index of the element that will be used to generate the proof and the *index* in the proof is the decimal representation of the reverse of the path.

If the number of elements in the path is equal to the depth of the tree, it means that all levels (except the root level) were included in the proof because they are necessary to calculate the root. In this case, the *index* in the proof will have the same value as the *index* in the function parameter.

However, if the path contains fewer elements than the depth of the tree, it means that some levels (except the root level) were omitted from the proof because they were unnecessary for calculating the root value. In this case, the two *index* values will differ.

3.6.2 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: $d + 1$

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

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

3.7 Verify Merkle Proof

Function to verify a Merkle Proof of a leaf in a LeanIMT.

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 calculated root matches the one that is part of the root, the algorithm will return true, otherwise it will return false.

Diagrams when verifying a Merkle Proof

Verifying a proof for node a_4

Note: This proof was generated using case 1 of the *generateProof* function.

path: [1]

Merkle Proof: {

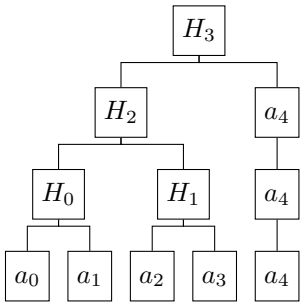
root: H_3

leaf: a_4

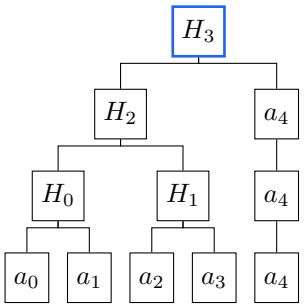
index: 1

siblings: $[H_2]$

}



LeanIMT to verify a proof for a_4



Nodes rebuilt to verify a proof for a_4

Verifying a proof for node a_3

Note: This proof was generated using case 2 of the *generateProof* function.

path: [1, 1, 0]

Merkle Proof: {

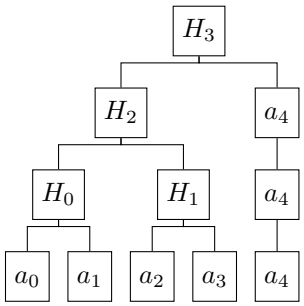
root: H_3

leaf: a_3

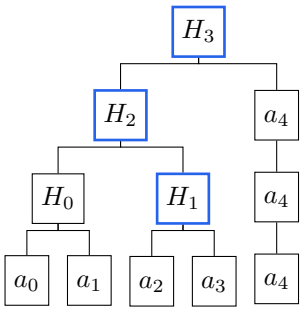
index: 3

siblings: $[a_2, H_0, a_4]$

}



LeanIMT to verify a proof for a_3



Nodes rebuilt to verify a proof for a_3

3.7.1 Pseudocode

Algorithm 6 LeanIMT verifyProof algorithm

```
1:  procedure VERIFYPROOF(index)
2:      { root, leaf, siblings, index } ← proof          ▷ Deconstruct the proof
3:      node ← leaf
4:      for i from 0 to siblings.length - 1 do
5:          isOdd ← divide index by 2 i times and check if the result is odd
6:          if isOdd is true then                        ▷ node is a right child
7:              node ← hash(siblings[i], node)
8:          else                                         ▷ It's a left node
9:              node ← hash(node, siblings[i])
10:         end if
11:     end for
12:     if root is equal node then
13:         return true
14:     else
15:         return false
16:     end if
17: end procedure
```

3.7.2 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 and then compare if the calculated root matches the root that is part of the Merkle Proof.

Number of operations to verify a Merkle Proof (worst case): $d + 1$

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

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

4 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 [15].

4.1 TypeScript/JavaScript

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

4.2 Solidity

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

5 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)

5.1 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

5.2 TypeScript/JavaScript

Note: The following TypeScript/JavaScript benchmarks demonstrate how to perform the same operations for each data structure. This means that even if the remove function of the IMT is called *Delete* in the implementation, it will be referred to as *Remove* in the benchmarks. Additionally, the IMT does not have a function called $InsertMany$, the $IMT - InsertMany$ benchmarks use the $Insert$ function in a loop to simulate inserting multiple members at the same time.

Note: Although the IMT does not have a batch insertion function, it does include an optimization for adding multiple members through its constructor.

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

For example:

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

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

5.2.1 Node.js

Table 1: All Functions in Node.js (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

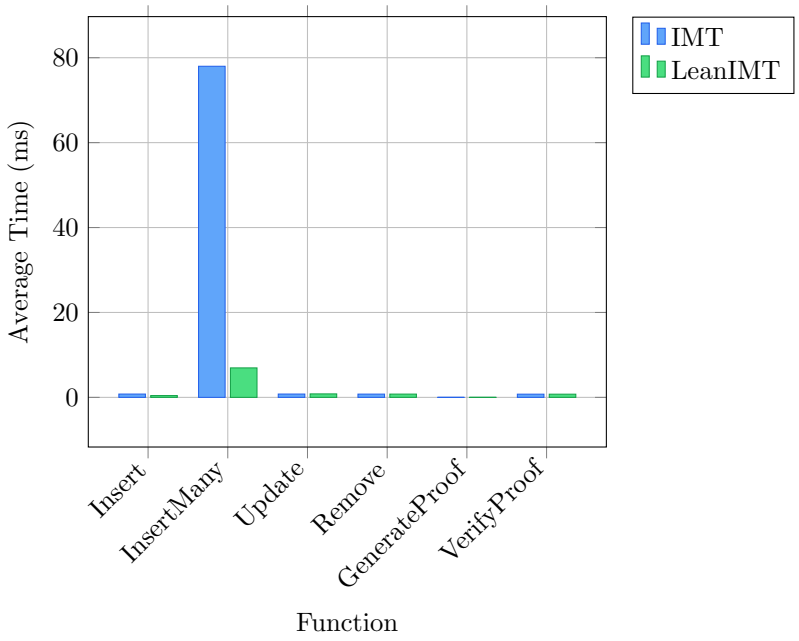


Figure 1: Functions IMT vs LeanIMT (100 iterations) in Node.js

Figure 2 shows the same information as Figure 1, but uses a logarithmic scale to better highlight the relationships between the functions.

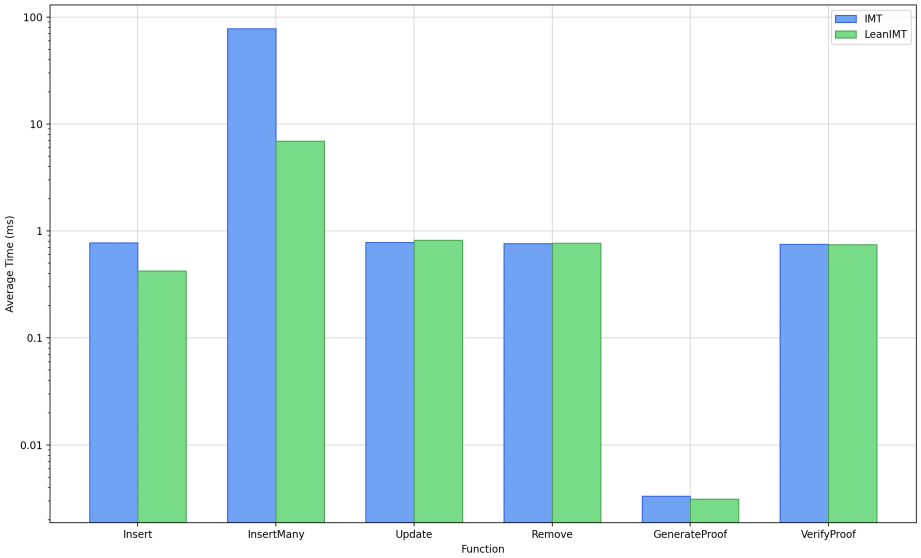


Figure 2: Functions IMT vs LeanIMT (100 iterations) in Node.js - Logarithmic scale

5.2.2 Browser

Table 2: All Functions in Browser (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

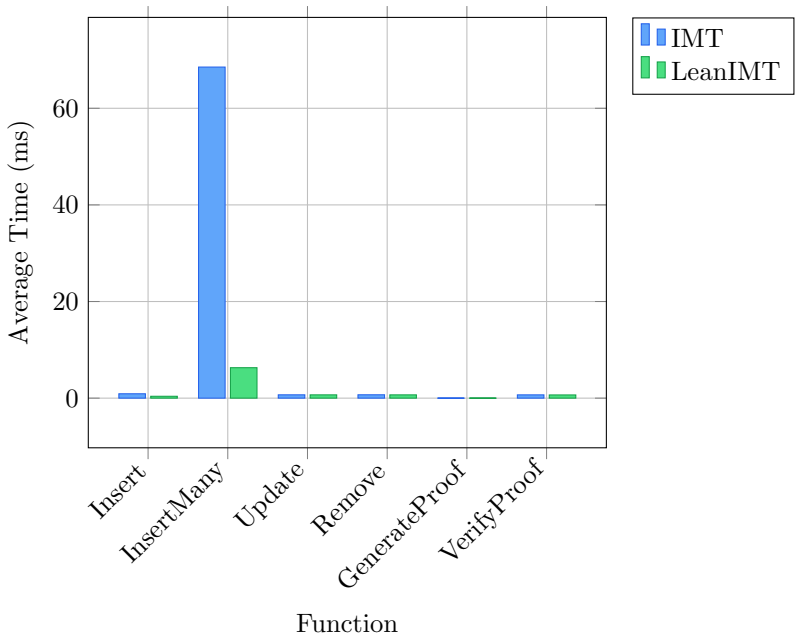


Figure 3: Functions IMT vs LeanIMT (100 iterations) in Browser

Figure 4 shows the same information as Figure 3, but uses a logarithmic scale to better highlight the relationships between the functions.

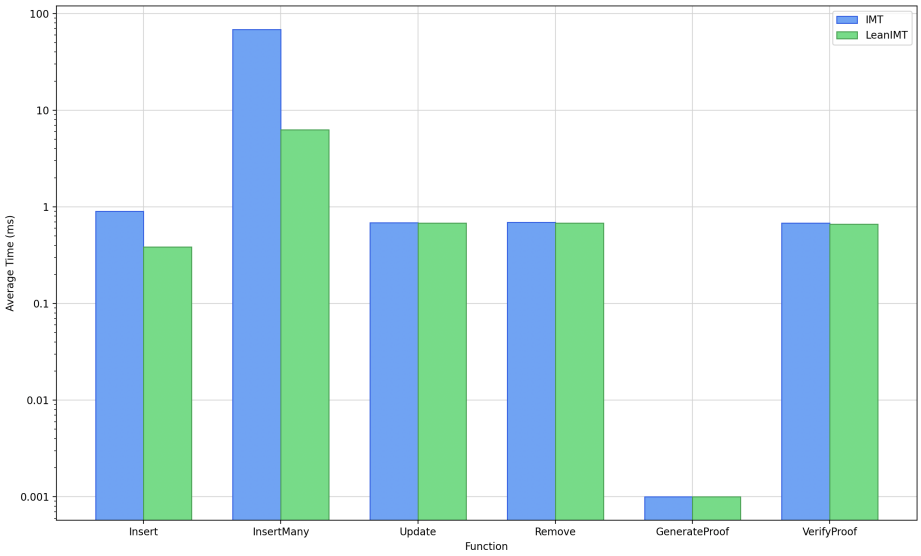


Figure 4: Functions IMT vs LeanIMT (100 iterations) in Browser - Logarithmic scale

5.2.3 LeanIMT: Node.js vs Browser

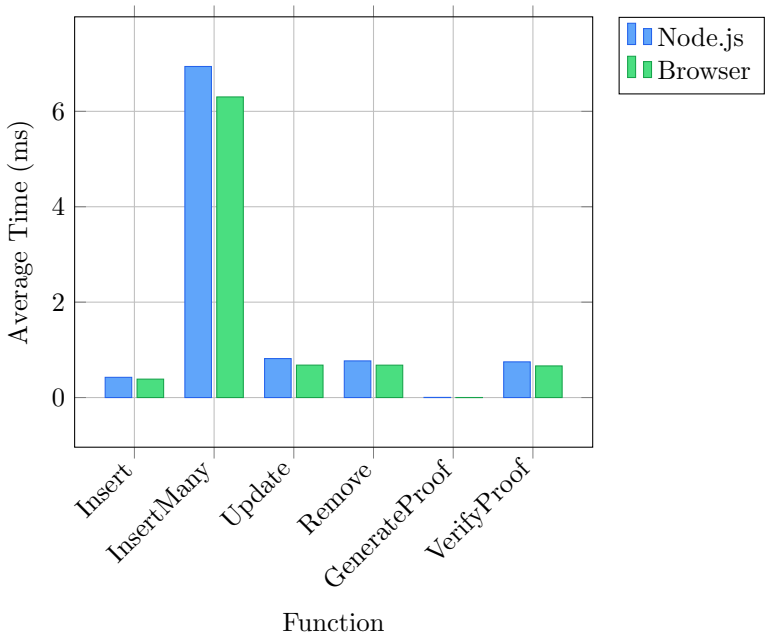


Figure 5: LeanIMT Node.js vs Browser (100 iterations)

5.2.4 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

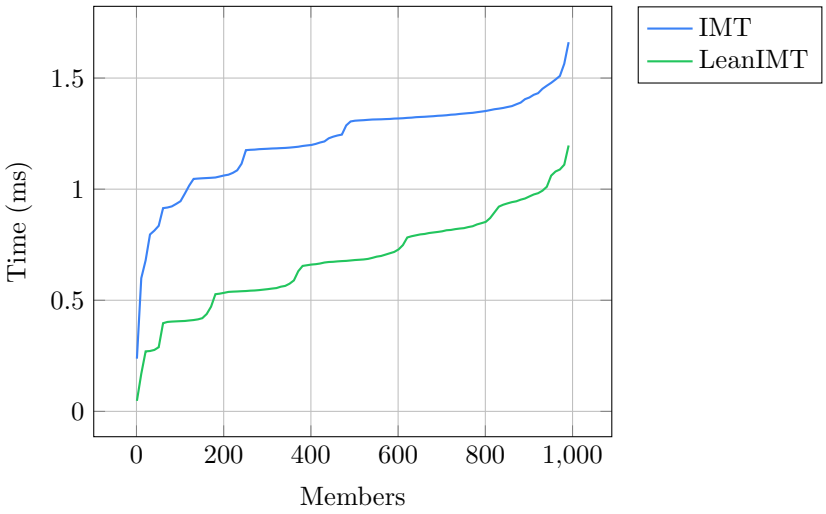


Figure 6: Insert function IMT vs LeanIMT (1000 iterations)

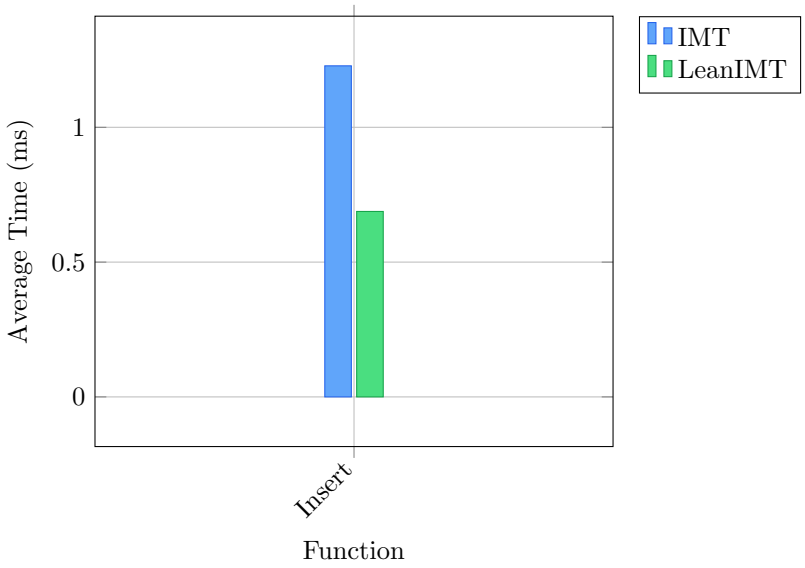


Figure 7: Insert function IMT vs LeanIMT (1000 iterations)

5.2.5 LeanIMT: Insert Loop vs Batch Insertion

Comparing the LeanIMT *Insert* function in a loop with the LeanIMT *InsertMany* function.

Table 4: Insert Function (100 iterations)

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

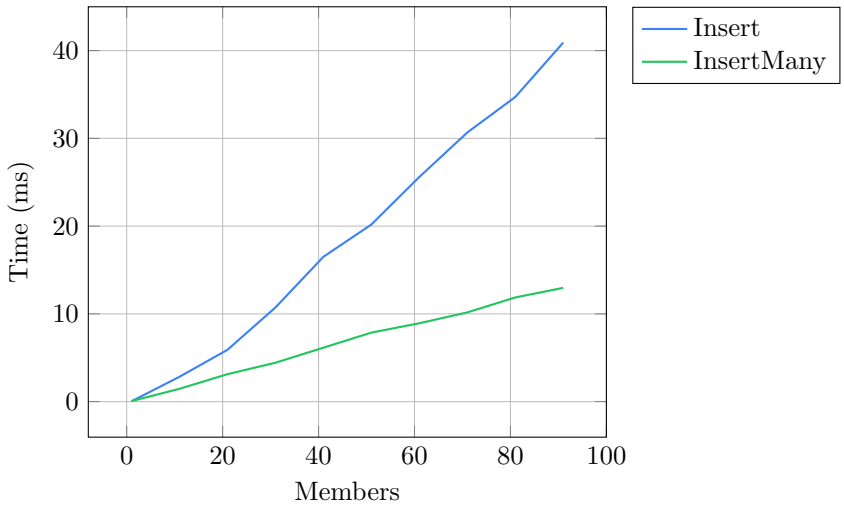


Figure 8: Batch Insertion LeanIMT (100 iterations)

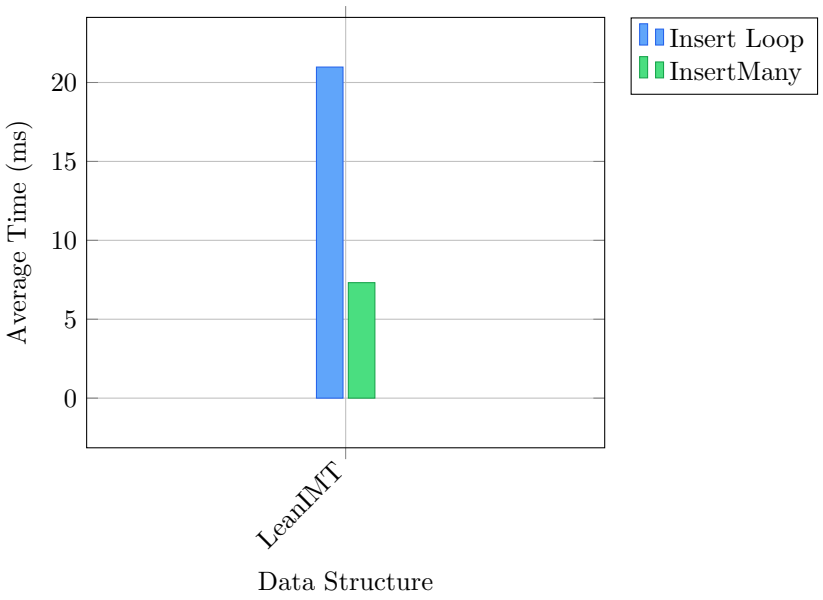


Figure 9: Batch Insertion LeanIMT (100 iterations)

5.3 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

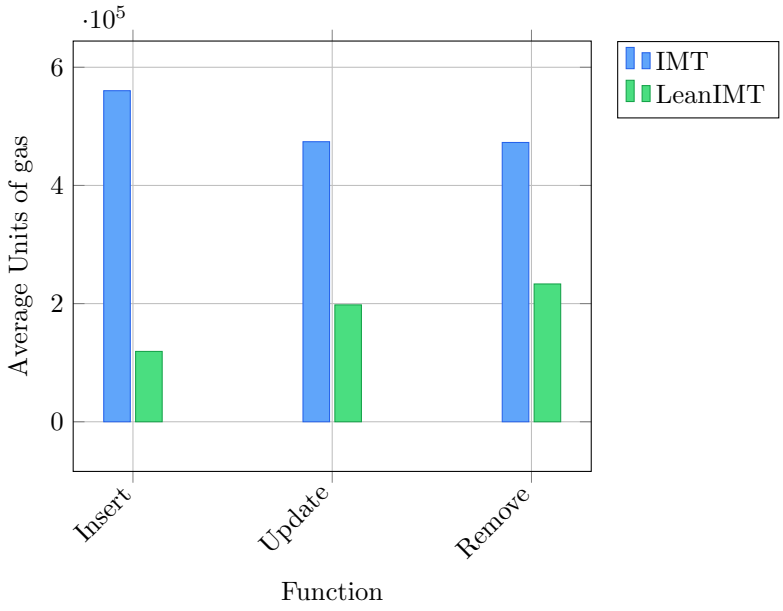


Figure 12: Gas cost of the execution of the Functions IMT vs LeanIMT

6 Conclusions

This technical document explains the LeanIMT algorithms and analyzes their time complexity. The benchmarks show the improvements of LeanIMT, which is the data structure used in Semaphore v4, over the IMT used in Semaphore v3.

6.1 Future Work

As future work, a function to update many members at once (similar to the *insertMany* function to insert many members at once) will be developed. Additionally, a Rust implementation of the data structure will be created to benchmark the performance in Node.js and browser environments.

References

[1] *IMT Solidity implementation*. URL: https://github.com/privacy-scaling-explorations/zk-kit.solidity/tree/main/packages/imt.

[2] *IMT TypeScript implementation*. URL: https://github.com/privacy-scaling-explorations/zk-kit/tree/main/packages/imt.

[3] *Incremental Merkle Tree*. URL: https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-16-zero-knowledge-zk/definitions-and-essentials/incremental-merkle-tree.

[4] G. Mohr J. Chapweske. "Tree Hash EXchange format (THEX)". In: (2003). URL: https://web.archive.org/web/20090803220648/http://open-content.net/specs/draft-jchapweske-thex-02.html.

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

[6] *LazyTower Solidity implementation*. URL: https://github.com/privacy-scaling-explorations/zk-kit.solidity/tree/main/packages/lazytower.

[7] *LazyTower TypeScript implementation*. URL: https://github.com/privacy-scaling-explorations/zk-kit/tree/main/packages/lazytower.

[8] LCamel. "LazyTower". In: (2023). URL: https://hackmd.io/@LCamel/LazyTower.

[9] *Merkle Mountain Range (MMR)*. URL: https://www.chainsecurity.com/blog/merkle-mountain-range-mmr-the-case-of-herodotus.

[10] *Merkle Mountain Ranges*. URL: https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md.

[11] *Merkle Mountain Ranges*. URL: https://github.com/mimblewimble/grin/blob/master/doc/mmr.md.

[12] *Merkle Mountain Ranges (MMR)*. URL: https://docs.grin.mw/wiki/chain-state/merkle-mountain-range.

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

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

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

[16] *Semaphore Website*. URL: https://semaphore.pse.dev.

[17] Alin Tomescu. "What is a Merkle Tree?" In: (2020). URL: https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree.

[18] *What are zero-knowledge proofs?* URL: https://ethereum.org/en/zero-knowledge-proofs.