Indexed Merkle Tree

Overview

This article will introduce the concept of an indexed merkle tree, and how it can be used to improve the performance of nullifier trees in circuits.

This page will answer:

- Why we need nullifier trees at all
- How indexed merkle trees work
- How they can be used for membership exclusion proofs
- How they can leverage batch insertions
- Tradeoffs of using indexed merkle trees

The content was also covered in a presentation for the Privacy + Scaling Explorations team at the Ethereum Foundation.

Primer on Nullifier Trees

Currently the only feasible way to get privacy in public blockchains is via a UTXO model. In this model, state is stored in encrypted UTXO's in merkle trees. However, to maintain privacy, state can not be updated. The very act of performing an update leaks information. In order to simulate "updating" the state, we "destroy" old UTXO's and create new ones for each state update. Resulting in a merkle tree that is append-only.

A classic merkle tree:

To destroy state, the concept of a "nullifier" tree is introduced. Typically represented as a sparse Merkle Tree, the structure is utilized to store notes deterministically linked to values in the append-only tree of encrypted notes.

A sparse merkle tree (not every leaf stores a value):

In order to spend / modify a note in the private state tree, one must create a nullifier for it, and prove that the nullifier does not already exist in the nullifier tree. As nullifier trees are modeled as sparse merkle trees, non membership checks are (conceptually) trivial.

Data is stored at the leaf index corresponding to its value. E.g. if I have a sparse tree that can contain 2^{256} 2^{256} 2^{256} values and want to prove non membership of the value 2^{128} 2^{128} 2^{128}. I can prove via a merkle membership proof that $tree\_values[2^{128}] = 0$ tree_values[2^{128}] = 0 $tree\_values[2^{128}]$

$= 0$, conversely if I can prove that $tree\_values[2^{128}] == 1$ tree_values[2^{128}] == 1 $tree\_values[2^{128}]$

$== 1$ I can prove that the item exists.

Problems introduced by using Sparse Merkle Trees for Nullifier Trees

While sparse Merkle Trees offer a simple and elegant solution, their implementation comes with some drawbacks. A sparse nullifier tree must have an index for $e \in \mathbb{F}_p$ e \in \mathbb{F}_p e

$\in \mathbb{F}_p$, which for the bn254 curve means that the sparse tree will need to have a depth of 254. If you're familiar with hashing in circuits the alarm bells have probably started ringing. A tree of depth 254 means 254 hashes per membership proof. In routine nullifier insertions, a non membership check for a value is performed, then an insertion of said value. This amounts to two trips from leaf to root, hashing all the way up. This means that there are $254 * 2$ 254*2 254

$* 2$ hashes per tree insertion. As the tree is sparse, insertions are random and must be performed in sequence. This means the number of hashes performed in the circuit scales linearly with the number of nullifiers being inserted. As a consequence number of constraints in a rollup circuit (where these checks are performed) will sky rocket, leading to long rollup proving times.

Indexed Merkle Tree Constructions

As it turns out, we can do better. This paper (page 6) introduces the idea of an indexed merkle tree. A Merkle Tree that allows us to perform efficient non-membership proofs. It achieves this by extending each node to include a specialized data structure. Each node not only stores some value $v \in \mathbb{F}_p$ v \in \mathbb{F}_p v

$\in \mathbb{F}_p$ but also some pointers to the leaf with the next higher value. The data structure is as follows:

leaf

{ v , i next , v next } . \textsf{leaf} = {v, i_{\textsf{next}}, v_{\textsf{next}}}. leaf

= { v ,

i next ,

v next } . Based on the tree's insertion rules, we can assume that there are no leaves in the tree that exist between the range(v , v next) (v, v_{\textsf{next}}) (v ,

v next) .

More simply put, the merkle tree pretty much becomes a linked list of increasing size, where once inserted the pointers at a leaf can change, but the nullifier value cannot.

Despite being a minor modification, the performance implications are massive. We no longer position leaves in place(i n d e x = = v a l u e) (index == value) (in d e x

== v a l u e) therefore we no longer need a deep tree, rather we can use an arbitrarily small tree (32 levels should suffice). Some quick back of the napkin maths can show that insertions can be improved by a factor of 8(256 / 32) (256 / 32) (256/32) .

For the remainder of this article I will refer to the node that provides the non membership check as a "low nullifier".

The insertion protocol is described below:

1. Look for a nullifier's corresponding low_nullifier where:
2. l
3. o
4. w
5. _
6. n
7. u
8. l
9. l
10. i
11. f
12. i
13. e
14. r
15. next_value
16.

17. v
18. low_nullifier_{\textsf{next_value}} > v
19. l
20. o
21. w
22. _
23. n
24. u
25. ll
26. i
27. f
28. i
29. e
30. r
31. next_value
32.
33.

34. v
35. ifn
36. e
37. w
38. _
39. n
40. u

41. l
42. l
43. i
44. f
45. i
46. e
47. r
48. new_nullifier
49. n
50. e
51. w
52. _
53. n
54. u
55. ll
56. i
57. f
58. i
59. er
60. is the largest use the leaf:
61. l
62. o
63. w
64. _
65. n
66. u
67. l
68. l
69. i
70. f
71. i
72. e
73. r
74. next_value
75. =
76. =
77. 0
78. low_nullifier_{\textsf{next_value}} == 0
79. l
80. o
81. w
82. _
83. n
84. u
85. ll
86. i
87. f
88. i
89. e
90. r
91. next_value
92.
93. ==
94. 0
95. Perform membership check of the low nullifier.
96. Perform a range check on the low nullifier's value and next_value fields:

n e w _ n u l l i f i e r

l o w _ n u l l i f i e r value

& &

(n e w _ n u l l i f i e r < l o w _ n u l l i f i e r next_value

∥

l o w _ n u l l i f i e r next_value = = 0) new_nullifier > low_nullifier_{\textsf{value}} \: \&\& \: (new_nullifier < low_nullifier_{\textsf{next_value}} \: \| \: low_nullifier_{\textsf{next_value}} == 0) n e w _ n u ll i f i er l o w _ n u ll i f i e r value

&&

(n e w _ n u ll i f i er

< l o w _ n u ll i f i e r next_value

‖

l o w _ n u ll i f i e r next_value

== 0) 1. Update the low nullifier pointers 2. l 3. o 4. w 5. _ 6. n 7. u 8. l 9. l 10. i 11. f 12. i 13. e 14. r 15. next_index 16. = 17. n 18. e 19. w 20. _ 21. i 22. n 23. s 24. e 25. r 26. t 27. i 28. o 29. n 30. _ 31. i 32. n 33. d 34. e 35. x 36. low_nullifier_{\textsf{next_index}} = new_insertion_index 37. l 38. o 39. w 40. _ 41. n 42. u 43. ll 44. i 45. f 46. i 47. e 48. r 49. next_index 50. 51. = 52. n 53. e 54. w 55. _ 56. in 57. ser 58. t 59. i 60. o 61. n 62. _ 63. in 64. d 65. e 66. x 67. l 68. o 69. w 70. _ 71. n 72. u 73. l 74. l 75. i 76. f 77. i 78. e 79. r 80. next_value 81. = 82. n 83. e 84. w 85. _ 86. n 87. u 88. l 89. l 90. i 91. f 92. i 93. e 94. r 95. low_nullifier_{\textsf{next_value}} = new_nullifier 96. l 97. o 98. w 99. _ 100. n 101. u 102. ll 103. i 104. f 105. i 106. e 107. r 108. next_value 109. 110. = 111. n 112. e 113. w 114. _ 115. n 116. u 117. ll 118. i 119. f 120. i 121. er 122. Perform insertion of new updated low nullifier (yields new root) 123. Update pointers on new leaf. Note: low_nullifier is from before update in step 4

n e w _ n u l l i f i e r _ l e a f value = n e w _ n u l l i f i e r new_nullifier_leaf_{\textsf{value}} = new_nullifier n e w _ n u ll i f i er _ l e a f value

= n e w _ n u ll i f i er n e w _ n u l l i f i e r _ l e a f next_value = l o w _ n u l l i f i e r next_value new_nullifier_leaf_{\textsf{next_value}} = low_nullifier_{\textsf{next_value}} n e w _ n u ll i f i er _ l e a f next_value

= l o w _ n u ll i f i e r next_value n e w _ n u l l i f i e r _ l e a f next_index = l o w _ n u l l i f i e r next_index new_nullifier_leaf_{\textsf{next_index}} = low_nullifier_{\textsf{next_index}} n e w _ n u ll i f i er _ l e a f next_index

= l o w _ n u ll i f i e r next_index 1. Perform insertion of new leaf (yields new root)

Number of insertion constraints, in total:

- 3n
- hashes of 2 field elements (wheren
- is the height of the tree).
- 3
- hashes of 3 field elements.
- 2
- range checks.
- A handful of equality constraints.

Special cases You'll notice at step 3 thel o w _ n u l l i f i e r next_value low_nullifier_{\textsf{next_value}} l o w _ n u ll i f i e r next_value can be 0. This is a special case as if a value is the max, it will not point to anything larger (as it does not exist). Instead it points to zero. By doing so we close the loop, so we are always inserting into a ring, if we could insert outside the ring we could cause a split.

A visual aid for insertion is presented below:

1. Initial state

2. Add a new valuev

3. =
4. 30
5. v=30
6. v
7. =

8. 30

9. Add a new valuev

10. =
11. 10
12. v=10
13. v
14. =

15. 10

16. Add a new valuev
17. =
18. 20
19. v=20
20. v
21. =
22. 20
23. Add a new valuev
24. =
25. 50
26. v=50
27. v
28. =
29. 50

By studying the transitions between each diagram you can see how the pointers are updated between each insertion.

A further implementation detail is that we assume the first 0 node is pre-populated. As a consequence, the first insertion into the tree will be made into the second index.

Non-membership proof

Suppose we want to show that the value20 doesn't exist in the tree. We just reveal the leaf which 'steps over'20 . I.e. the leaf whose value is less than20 , but whose next value is greater than20 . Call this leaf thelow_nullifier .

- hash the low nullifier:l
- o
- w
- _
- n
- u
- l
- l
- i
- f
- i
- e
- r
- =
- h
- (
- 10
- ,
- 1
- ,
- 30
-)
- low_nullifier = h(10, 1, 30)
- l
- o
- w
- _
- n
- u
- ll
- i
- f
- i
- er
- =
- h
- (
- 10
- ,

- 1
- ,
- 30
-)
- .
- Prove the low leaf exists in the tree:n
- hashes.
- Check the new value 'would have' belonged in the range given by the low leaf:2
- range checks.* If (l
-
    - o
-
    - w
-
    - _
-
    - n
-
    - u
-
    - l
-
    - l
-
    - i
-
    - f
-
    - i
-
    - e
-
    - r
-
    - next_index
-
    - =
-
    - =
-
    - 0
-
    - low_nullifier_{\textsf{next_index}} == 0
-
    - l
-
    - o
-
    - w
-
    - _
-
    - n
-
    - u
-
    - ll
-
    - i
-
    - f
-
    - i
-
    - e
-
    - r
-

  - next_index
-
  -
-
  - ==
-
  - 0
-
  -):* Special case, the low leaf is at the very end, so the new_value must be higher than all values in the tree:
-
  -
    - a
-
  -
    - s
-
  -
    - s
-
  -
    - e
-
  -
    - r
-
  -
    - t
-
  -
    - (
-
  -
    - l
-
  -
    - o
-
  -
    - w
-
  -
    - _
-
  -
    - n
-
  -
    - u
-
  -
    - l
-
  -
    - l
-
  -
    - i
-
  -
    - f
-
  -
    - i
-
  -
    - e
-
  -

    - r
-
  -
    - value
-
  -
    - <
-
  -
    - n
-
  -
    - e
-
  -
    - w
-
  -
    - _
-
  -
    - v
-
  -
    - a
-
  -
    - l
-
  -
    - u
-
  -
    - e
-
  -
    - value
-
  -
    -)
-
  -
    - assert(low_nullifier_{\textsf{value}} < new_value_{\textsf{value}})
-
  -
    - a
-
  -
    - sser
-
  -
    - t
-
  -
    - (
-
  -
    - l
-
  -
    - o
-
  -
    - w
-
  -
    - _
-

  -
    - n
-
  -
    - u
-
  -
    - ll
-
  -
    - i
-
  -
    - f
-
  -
    - i
-
  -
    - e
-
  -
    - r
-
  -
    - value
-
  -
    -
-
  -
    - <
-
  -
    - n
-
  -
    - e
-
  -
    - w
-
  -
    - _
-
  -
    - v
-
  -
    - a
-
  -
    - l
-
  -
    - u
-
  -
    - e
-
  -
    - value
-
  -
    -
-
  -
    -)

-
    - Else:* a
-
    -
        - s
-
    -
        - s
-
    -
        - e
-
    -
        - r
-
    -
        - t
-
    -
        - (
-
    -
        - l
-
    -
        - o
-
    -
        - w
-
    -
        - _
-
    -
        - n
-
    -
        - u
-
    -
        - l
-
    -
        - l
-
    -
        - i
-
    -
        - f
-
    -
        - i
-
    -
        - e
-
    -
        - r
-
    -
        - value
-
    -
        - <
-
    -
        - n

-
  -
    - e
-
  -
    - w
-
  -
    - _
-
  -
    - v
-
  -
    - a
-
  -
    - l
-
  -
    - u
-
  -
    - e
-
  -
    - value
-
  -
    -)
-
  -
    - assert(low_nullifier_{\textsf{value}} < new_value_{\textsf{value}})
-
  -
    - a
-
  -
    - sser
-
  -
    - t
-
  -
    - (
-
  -
    - l
-
  -
    - o
-
  -
    - w
-
  -
    - _
-
  -
    - n
-
  -
    - u
-
  -
    - ll
-
  -

    - i
-
  -
    - f
-
  -
    - i
-
  -
    - e
-
  -
    - r
-
  -
    - value
-
  -
    -
-
  -
    - <
-
  -
    - n
-
  -
    - e
-
  -
    - w
-
  -
    - _
-
  -
    - v
-
  -
    - a
-
  -
    - l
-
  -
    - u
-
  -
    - e
-
  -
    - value
-
  -
    -
-
  -
    -)
-
  -
    - a
-
  -
    - s
-
  -
    - s
-

  -
    - e
-
  -
    - r
-
  -
    - t
-
  -
    - (
-
  -
    - l
-
  -
    - o
-
  -
    - w
-
  -
    - _
-
  -
    - n
-
  -
    - u
-
  -
    - l
-
  -
    - l
-
  -
    - i
-
  -
    - f
-
  -
    - i
-
  -
    - e
-
  -
    - r
-
  -
    - next_value
-
  -
    -
-
  -
    - n
-
  -
    - e
-
  -
    - w
-
  -
    - _

-
  -
    - v
-
  -
    - a
-
  -
    - l
-
  -
    - u
-
  -
    - e
-
  -
    - value
-
  -
    -)
-
  -
    - assert(low_nullifier_{\textsf{next_value}} > new_value_{\textsf{value}})
-
  -
    - a
-
  -
    - sser
-
  -
    - t
-
  -
    - (
-
  -
    - l
-
  -
    - o
-
  -
    - w
-
  -
    - _
-
  -
    - n
-
  -
    - u
-
  -
    - ll
-
  -
    - i
-
  -
    - f
-
  -
    - i
-
  -

        - e
-
    -
        - r
-
    -
        - next_value
-
    -
        -
-
    -
        -
-
    -
        - n
-
    -
        - e
-
    -
        - w
-
    -
        - _
-
    -
        - v
-
    -
        - a
-
    -
        - l
-
    -
        - u
-
    -
        - e
-
    -
        - value
-
    -
        -
-
    -
        -)

This is already a massive performance improvement, however we can go further, as this tree is not sparse. We can perform batch insertions.

Batch insertions

As our nullifiers will all be inserted deterministically (append only), we can insert entire subtrees into our tree rather than appending nodes one by one, this optimization is globally applied to append only merkle trees. I wish this was it, but for every node we insert, we must also update low nullifier pointers, this introduces a bit of complexity while performing subtree insertions, as the low nullifier itself may exist within the subtree we are inserting - we must be careful how we prove these sort of insertions are correct (addressed later). We must update all of the impacted low nullifiers before.

First we will go over batch insertions in an append only merkle tree.

1. First we prove that the subtree we are inserting into consists of all empty values.
2. We work out the root of an empty subtree, and perform an inclusion proof for an empty root, which proves that there is nothing within our subtree.
3. We re-create our subtree within our circuit.
4. We then use the same sibling path the get the new root of the tree after we insert the subtree.

In the following example we insert a subtree of size 4 into our tree at step 4. above. Our subtree is greyed out as it is "pending".

Legend :

- Green: New Inserted Value
- Orange: Low Nullifier

Example

1. Prepare to insert subtree[
2. 35
3. ,
4. 50
5. ,
6. 60
7. ,
8. 15
9.]
10. [35,50,60,15]
11. [
12. 35
13. ,
14. 50
15. ,
16. 60
17. ,
18. 15
19.]
20. Update low nullifier for new nullifier35
21. 35
22. 35
23. .
24. Update low nullifier for new nullifier50
25. 50
26. 50
27. . (Notice how the low nullifier exists within our pending insertion subtree, this becomes important later).
28. Update low nullifier for new nullifier60
29. 60
30. 60
31. .
32. Update low nullifier for new nullifier15
33. 15
34. 15
35. .
36. Update pointers for new nullifier15
37. 15
38. 15
39. .
40. Insert subtree.

Performance gains from subtree insertion

Lets go back over the numbers: Insertions into a sparse nullifier tree involve 1 non membership check (254 hashes) and 1 insertion (254 hashes). If we were performing insertion for 4 values that would entail 2032 hashes. In the depth 32 indexed tree construction, each subtree insertion costs 1 non membership check (32 hashes), 1 pointer update (32 hashes) for each value as well as the cost of constructing and inserting a subtree (~67 hashes. Which is 327 hashes, an incredible efficiency increase.)

I am ignoring range check constraint costs as they a negligible compared to the costs of a hash .

Performing subtree insertions in a circuit context

Some fun engineering problems occur when we inserting a subtree in circuits when the low nullifier for a value exists within the subtree we are inserting. In this case we cannot perform a non membership check against the root of the tree, as our leaf that we would use for non membership has NOT yet been inserted into the tree. We need another protocol to handle such cases, we like to call these "pending" insertions.

Circuit Inputs

- new_nullifiers
- :fr[]
- low_nullifier_leaf_preimages
- :tuple of {value: fr, next_index: fr, next_value: fr}
- low_nullifier_membership_witnesses
- : A sibling path and a leaf index of low nullifier
- current_nullifier_tree_root
- : Current root of the nullifier tree
- next_insertion_index
- : fr, the tip our nullifier tree
- subtree_insertion_sibling_path
- : A sibling path to check our subtree against the root

Protocol without batched insertion: Before adding a nullifier to the pending insertion tree, we check for its non membership using the previously defined protocol by consuming the circuit inputs: Pseudocode:

auto empty_subtree_hash = SOME_CONSTANT_EMPTY_SUBTREE ; auto pending_insertion_subtree =

[ ] ; auto insertion_index = inputs . next_insertion_index ; auto root = inputs . current_nullifier_tree_root ;

// Check nothing exists where we would insert our subtree assert (membership_check (root , empty_subtree_hash , insertion_index

    subtree_depth , inputs . subtree_insertion_sibling_path)) ;

for

(i in len (new_nullifiers))

{ auto new_nullifier = inputs . new_nullifiers [i] ; auto low_nullifier_leaf_preimage = inputs . low_nullifier_leaf_preimages [i] ; auto low_nullifier_membership_witness = inputs . low_nullifier_membership_witnesses [i] ;

// Membership check for low nullifier assert (perform_membership_check (root ,

hash (low_nullifier_leaf_preimage) , low_nullifier_membership_witness)) ;

// Range check low nullifier against new nullifier assert (new_nullifier < low_nullifier_leaf_preimage . next_value || low_nullifier_leaf . next_value ==

0) ; assert (new_nullifier

  low_nullifier_leaf_preimage . value) ;

// Update new nullifier pointers auto new_nullifier_leaf =

{ . value = new_nullifier , . next_index = low_nullifier_preimage . next_index , . next_value = low_nullifier_preimage . next_value } ;

// Update low nullifier pointers low_nullifier_preimage . next_index = next_insertion_index ; low_nullifier_preimage . next_value = new_nullifier ;

// Update state vals for next iteration root =

update_low_nullifier (low_nullifier , low_nullifier_membership_witness) ; pending_insertion_subtree . push (new_nullifier_leaf) ; next_insertion_index +=

```
1 ; }

// insert subtree root =

insert_subtree ( root , inputs . next_insertion_index

          subtree_depth , pending_insertion_subtree ) ;
```

From looking at the code above we can probably deduce why we need pending insertion. If the low nullifier does not yet exist in the tree, all of our membership checks will fail, we cannot produce a non membership proof.

To perform batched insertions, our circuit must keep track of all values that are pending insertion.

- If thelow_nullifier_membership_witness
- is identified to be nonsense (all zeros, or has a leaf index of -1) we will know that this is a pending low nullifier read request and we will have to look within our pending subtree for the nearest low nullifier. *Loop back through all "pending_insertions"* If the pending insertion value is lower than the nullifier we are trying to insert
-
    -
        - If the pending insertion value is NOT found, then out circuit is invalid and should self abort.

The updated pseudocode is as follows:

```
auto empty_subtree_hash = SOME_CONSTANT_EMPTY_SUBTREE ; auto pending_insertion_subtree =

[ ] ; auto insertion_index = inputs . next_insertion_index ; auto root = inputs . current_nullifier_tree_root ;

// Check nothing exists where we would insert our subtree assert ( membership_check ( root , empty_subtree_hash ,
insertion_index

          subtree_depth , inputs . subtree_insertion_sibling_path ) ) ;

for

( i in len ( new_nullifiers ) )

{ auto new_nullifier = inputs . new_nullifiers [ i ] ; auto low_nullifier_leaf_preimage = inputs . low_nullifier_leaf_preimages [ i ]
; auto low_nullifier_membership_witness = inputs . low_nullifier_membership_witnesses [ i ] ;

if

( low_nullifier_membership_witness is garbage )

{ bool matched =

false ;

// Search for the low nullifier within our pending insertion subtree for

( j in range ( 0 , i ) )

{ auto pending_nullifier = pending_insertion_subtree [ j ] ;

if

( pending_nullifier . is_garbage ( ) )

continue ; if

( pending_nullifier [ j ] . value < new_nullifier &&

( pending_nullifier [ j ] . next_value

     new_nullifier || pending_nullifier [ j ] . next_value ==

0 ) )

{

// bingo matched =

true ;

// Update pointers auto new_nullifier_leaf =
{ . value = new_nullifier , . next_index = pending_nullifier . next_index , . next_value = pending_nullifier . next_value }

// Update pending subtree pending_nullifier . next_index = insertion_index ; pending_nullifier . next_value = new_nullifier ;

pending_insertion_subtree . push ( new_nullifier_leaf ) ; break ; } }

// could not find a matching low nullifier in the pending insertion subtree assert ( matched ) ;

}

else

{ // Membership check for low nullifier assert ( perform_membership_check ( root ,

hash ( low_nullifier_leaf_preimage ) , low_nullifier_membership_witness ) ) ;

// Range check low nullifier against new nullifier assert ( new_nullifier < low_nullifier_leaf_preimage . next_value ||
low_nullifier_leaf . next_value ==

0 ) ; assert ( new_nullifier

      low_nullifier_leaf_preimage . value ) ;

// Update new nullifier pointers auto new_nullifier_leaf =

{ . value = new_nullifier , . next_index = low_nullifier_preimage . next_index , . next_value = low_nullifier_preimage .
next_value } ;

// Update low nullifier pointers low_nullifier_preimage . next_index = next_insertion_index ; low_nullifier_preimage .
next_value = new_nullifier ;

// Update state vals for next iteration root =

update_low_nullifier ( low_nullifier , low_nullifier_membership_witness ) ;

pending_insertion_subtree . push ( new_nullifier_leaf ) ;

}

next_insertion_index +=

1 ; }

// insert subtree root =

insert_subtree ( root , inputs . next_insertion_index

            subtree_depth , pending_insertion_subtree ) ;
```

Drawbacks

Despite offering large performance improvements within the circuits, these come at the expense of increased computation / storage performed by the node. To provide a non membership proof we must find the "low nullifier" for it. In a naive implementation this entails a brute force search against the existing nodes in the tree. This performance can be increased by the node maintaining a sorted data structure of existing nullifiers, increasing its storage footprint.

Closing Notes

We have been working with these new trees in order to reduce the proving time for our rollups in Aztec, however we think EVERY protocol leveraging nullifier trees should know about these trees as their performance benefit is considerable. Edit this page

Previous Trees Next Storage Slots