Some zk-SNARK enabled applications built on Ethereum, such asSemaphore and Tornado Cash, use incremental binary Merkle trees to accumulate identity commitments, so users can prove set membership in zero-knowledge. Such applications, however, face two tradeoffs: first, between tree capacity and gas costs for tree initialisation and insertions; and second, between tree capacity and zk-SNARK circuit size. The former impacts users as they must spend more gas to insert a value into the tree (for instance, to deposit into a mixer). The latter impacts those who need to generate zero-knowledge proofs of set membership (for instance, to withdraw funds from a mixer).

Developers must therefore be very careful to select a suitable tree depth and hash function for their use case in order to minimise the impact on usability.

In this post, I present the gas consumption and circuit constraints of two snark-friendly hash functions applied to Merkle trees of various depths, and show that with the Poseidon hash function, we can maximise tree capacity with quinary Merkle trees (5 leaves per node) and save on gas costs with binary Poseidon Merkle trees. Furthermore, assuming that Poseidon has equal or greater cryptographic security than MiMC, developers should use Poseidon for Merkle trees, regardless of the number of nodes per leaf.

Note that the cryptographic security of the MiMC and Poseidon hash functions is out of scope of this post.

Background

The deeper a Merkle tree, the more gas is required to initialise and append elements to it. Moreover, the number of constraints required by a zk-SNARK circuit to verify a Merkle proof increases with the tree depth. In turn, this increases proving time, proving key size, and circuit file size.

A common practice is to reduce the number of constraints for tree verification circuits is use a snark-friendly hash function to generate commitments. A snark-friendly hash function is named as such as it is particularly efficient inside a zk-SNARK circuit and requires a low number of circuit constraints, as opposed to hash functions whose operation requires many constraints. The canonical SHA256 hash function, for instance, may be efficient outside of a zk-SNARK circuit, but is highly inefficient inside zk-SNARKs. Using a snark-friendly hash function significantly reduces the number of circuit constraints and proof generation time, but on-chain hashes require more gas.

MiMC and Poseidon are two such hash functions. There are others, but this post focuses on them as working implementations exist in the the [circomlib

](https://github.com/iden3/circomlib) library.

About incremental Merkle trees

An incremental Merkle tree is optimised for efficient appends, but does not support leaf updates. When implemented as an Ethereum contract, it maintains the following storage variables:

1. The Merkle root
2. The zero-leaf value per tree level (e.g. [0, hash(0, 0), hash(hash(0, 0), hash(0, 0))...]

)

1. Cached values required for efficient appends, also known as filled subtrees

An audited implementation of a binary incremental Merkle tree is availablehere.

Gas costs for MiMC and Poseidon

circomlib

provides MiMC and Poseidon implementations as directly-generated EVM bytecode. To hash two uint256

values on-chain requires the following gas:

Hash function

Gas cost to hash 2 values

Notes

MiMC

59840

Using MiMCSponge as implemented here.

Poseidon

49858

Using t = 3

. See the implementation here.

SHA256

23179

For reference only.

We find that the Poseidon hash function uses less gas than MiMC to hash the same number of values in a single function call.

MiMC

The gas cost required to append a value to an incremental Merkle tree is linear with its depth. The same can be said for tree initialisation, which involves hashing and caching the zero leaf per level.

If we use MiMC in a sponge construction to create and insert values into a binary incremental Merkle tree, the gas costs and circuit constraints are as follows:

Depth

Capacity

Initialisation gas cost

Insertion gas cost

Path verification circuit constraints

10

1,024

1,289,360

528,298

13,230

20

1,048,576

2,075,640

969,498

26,460

30

107,3741,824

2,861,920

1,410,634

39,690

Poseidon

Since PoseidonT3 can hash 2 elements, we can use it for an incremental binary Merkle tree. The associated gas costs and zk-SNARK constraints for path verification are as follows:

Depth

Capacity

Initialisation gas cost

Insertion gas cost

Path verification circuit constraints

10

1,024

1,317,741

427,238

2,190

20

1,048,576

2,003,430

767,565

4,380

30

1,073,741,824

2,689,180

1,107,895

6,570

Note that the Poseidon hash function accepts the following parameters: a seed, t

, roundFull

, and roundPartial

. The value of t

should be the number of elements to hash, plus one for the reasons described in this document.

Name

t

roundFull

roundPartial

seed

PoseidonT3

3

8

49

“poseidon”

PoseidonT6

6

8

50

“poseidon”

Due to limitations with the EVM and the circomlib

implementation of the Poseidon contract generator, the maximum t

value for an on-chain Poseidon contract that circomlib

can generate is 6, so the largest-capacity Poseidon hash function contract can accept 5 values.

Hashing 5 values using PoseidonT6 requires 109166 gas. The average amount of gas per input value (109166 / 5 = 21833

) is less than that of PoseidonT3 (49858 / 2 = 24929

).

Quinary Merkle trees

It would be a waste to use PoseidonT6 to only hash 2 values per tree level. We can use PoseidonT6 to the fullest with an incremental Merkle tree that supports 5 leaves per node, also known as a quinary Merkle tree.

Depth

Capacity

Compare to

Initialisation gas cost

Insertion gas cost

Path verification circuit constraints

5

3,125

210

1,747,294

659,869

2,182

8

390,625

2,375,257

1,012,441

3,528

9

1,953,125

220

2,584,578

1,129,965

3,969

10

9,765,625

2,793,899

1,247,425

4,410

13

1,220,703,125

230

3,421,862

1,600,061

5,733

16

152,587,890,625

4,049,825

1,952,569

7,056

20

95,367,431,640,625

4,887,109

2,422,729

8,820

30

931,322,574,615,478,515,625

6,980,319

3,597,969

13,230

Note that in order to make a meaningful comparision of gas costs, it is important to do so on the capacity of the tree, not its depth, as the number of leaves per node is different. As such, it is meaningful to compare a quinary tree of depth 9 to a binary tree of depth 20.

The biggest advantage of using a a quinary Poseidon Merkle tree is that it requires very few constraints, and that it can hold orders of magnitude more leaves than its binary counterparts. The downside is that its gas consumption is higher.

Quinary Poseidon Merkle trees for MACI

In particular, Minimum Anti-Collusion Infrastructure (MACI) can benefit from quinary trees.

MACI uses a zk-SNARK to tally votes. This circuit accepts a list of results (one value per vote option) and outputs a commitment to the results. To generate this commitment, it accumulates all the results into a Merkle root (and later hashes it with a salt).

The circuit which accumulates a list of leaves into a single root (using a Poseidon binary Merkle tree) is implementedhere, and the equivalent circuit which uses Poseidon quinary tree is implemented here.

With a binary Merkle tree of depth n

, the circuit needs to perform (2 ^ n) - 1

hashes. With a quinary Merkle tree of depth m

, the circuit needs to perform (5 ^ m) - 1

hashes.

With a Poseidon binary Merkle tree:

Depth

Capacity

Constraints

4

16

3,240

5

32

6696

7

128

27432

9

512

110,376

With a Poseidon quinary Merkle tree:

Depth

Capacity

Compare to

Constraints

2

25

2 4

1,746

3

125

25

, and 27

or 28

9,021

4

625

29

45,396

Assuming that we need MACI to support at most 512 vote options, we can achieve a 58% reduction in constraints for this circuit if we use a Poseidon quinary Merkle tree with 4 levels, rather than a binary Merkle tree of 9 levels. In MACI, only the coordinator needs to generate vote tally proofs, and since result tally verification does not involve any tree insertions, there are no tradeoffs if we use a Poseidon quinary Merkle trees for this circuit.

The same cannot be said, however, of using Poseidon quinary Merkle trees for MACI's state tree and message tree, since users may be senstitive to insertion gas costs. If they are, it is absolutely acceptable to use Poseidon binary Merkle trees for those trees.

Conclusion

Ultimately, whether the gas tradeoff is worth it depends on one's use case. For instance, for an application that requires a tree that supports between 1 - 2 million leaves, a a binary Merkle tree that uses Poseidon is favourable, as it requires at least 767k gas per insertion, rather than 969k gas per insertion for a MiMC binary Merkle tree, or 1129k gas per insertion for a Poseidon quinary Merkle tree.

If the use case, however, requires efficient proof generation and is less sensitive to gas costs, a quinary Merkle tree is an excellent choice (as it uses 3969 constraints vs 4380 for a Poseidon binary Merkle tree and 26460 for a MiMC binary Merkle tree).

Tree type

Capacity

Gas consumption

Proof generation efficiency

MiMC binary

1

2

1

Poseidon binary

1

3

2

Poseidon quinary

3

1

3

(Note: 1 means least ideal, and 3 means most ideal.)

Finally, for scenarios such as vote tally proof generation in MACI, Poseidon quinary Merkle trees are the best option and do not impose any gas consumption tradeoffs.

Credits

Many thanks to @liangcc for implementing the Poseidon hash functions for Minimal Anti-Collusion Infrastructure, @jbaylina and the team at iden3 for their work on circomlib

, and @kobigurk for his advice and feedback.