Shorter Merkle proofs for Snapps

Resembling construction was published by farazhaider for Plasma some days ago. Thank you.

Let's we represent the way to reduce Merkle proofs for Snapps.

Abstract

Below we will describe a way to reduce a proof size given by aMerkle tree. The way is based on a special - balanced approach to build the tree of hashes. The main idea is that keeping a balanced tree of hashes ensures fewer nodes count between the root and any element even in the worst possible case, so that makes a proof shorter.

Merkle trees

As a good and allied way to store data in zkSnarks is asparse Merkle tree. The main idea of such tree is keeping the nodes for the whole space of all possible values. There are ways to optimize it on the implementation stage, but the proof part (the proof of including) seems to be too long and may take 160 or more hashes that are complicated to deal with.

This number can be reduced by choosing a different way to build the Merkle tree. As another possible way, we could choose a simple binary tree. Averagely, it has the depth depending on the number of elements, that is better. But the worst case may be even worse than in sparse Merkle tree with respect to the proof size that is still too expensive in use.

In order to resolve the problem of the worst case, we add balancing to the tree building process. As an approach, we chose an AVL tree as one of a possible and quite simple way to implement and use.

AVL tree approach

AVL tree is a binary tree that changes its structure after adding or removing an element, in order to keep the total depth (the maximal distance between the root and a node) as less as possible. On the picture, there are examples of balanced and unbalanced trees.

In AVL tree the balance is reached by two types of rotations in the nodes. For example, if an element is added (or removed), we check each node in the path to the element for the difference in depths of its left and right sub-trees. If the difference is 2 we make a rotation according to the rules invented and discovered by Adelson-Velsky and Landis (this is why such tree is called AVL). There four types of rotations:

After the correct rotation for the node, its depths difference will be 0 or 1, thus the tree will be balanced.

Ordered balanced Merkle tree

Supposing, we use AVL approach to build the Merkle tree.

Let's demand following properties of our construction:

- the difference between heights of left and right subtrees is equal {-1, 0, 1}
- each leaf has a unique ID and left leaf's ID is lesser then right.

$N(h)$

is the capacity of the state in the worst case.

It is obvious, that $N(1)=1$

and $N(2)=2$

.

Also, $N(h)=N(h-1)+N(h-2)$

- it is maximal unbalanced.

We see that $N(h) = F_{h+1}$

- it is Fibbonaci number.

$h \leq [0.69 \log_{2} (N \sqrt{5} + \frac{1}{2})] - 1$

Talking about the sparse Merkle tree, the proof size does not depend on the number of elements, it depends on the size of the values' space. We can imagine sparse tree as a fully balanced tree that has almost all leaves (places for elements) empty, this is why it is called sparse. In spite of it, ordered a balanced Merkle tree keeps only leaves with inserted elements. Obviously, that's why the sparse Merkle tree has longer proof size.

For 160bit address space the length for Merkle proof for SMT is 160 and length for OBMT it is 58 (for 2^40 memory cells) and 44 (for 2^30 memory celles).

This result may be useful for snapps, allowing to reduce the number of constraints in 3-4 times.

UX for snapps using OBMT

Snapps based on OBMT memory must have two different kinds of state modification:

1. memory defragmentation
2. properly, snapp execution

At memory defragmentation step we can add or remove cells with zero value and rotate the OBMT.

Snapp can be split into two snarks. Then we can combine several execution blocks of the snapp with one defragmentation block.

We represent circuit for defragmentation gadget in pseudocode (unoptimized at the low level, just a draft).

external constant MAX_HEIGHT

struct Rotation uint2 type uint160 [4] nodes end

struct Proof uint160 [MAX_HEIGHT] data bits[MAX_HEIGHT] way bits[MAX_HEIGHT] included end

struct Item uint160 id uint160 data end

gadget insertRightAndRotate(Proof proof, uint160 leaf1, uint160 leaf2, Rotation rotation) returns(uint160): let rotationTo := hash(hash(rotation.nodes[1],rotation.nodes[2]), hash(rotation.nodes[3], rotation.nodes[4])) let rotationFrom := case4(rotation.type, hash(hash(hash(rotation.nodes[1],rotation.nodes[2]), rotation.nodes[3]), rotation.nodes[4]), hash(hash(rotation.nodes[1], hash(rotation.nodes[2], rotation.nodes[3])), rotation.nodes[4]), hash(rotation.nodes[1], hash(hash(rotation.nodes[2], rotation.nodes[3]), rotation.nodes[4])), hash(rotation.nodes[1], hash(rotation.nodes[2], hash(rotation.nodes[3], rotation.nodes[4])))) let top := hash(leaf1, leaf2) for i = MAX_HEIGHT .. 1: let _top := orderedHash(proof.way[i], proof.data[i], top) let __top := rotationTo if _top == rotationFrom else _top let top := case2(proof.included[i], top, __top) end return top end

gadget addLeft(uint160 rootFrom, uint160 rootTo, uint160 id, Proof pleft, Item ileft, Proof pright, Item iright, Rotation rotation): constraints uint(pleft.way) == uint(pright.way)-1 constraints merkleProof(pleft, hash(ileft)) == rootFrom constraints merkleProof(pright, hash(iright)) == rootFrom constraints ileft.id < id constraints id < iright.id constraints rootTo == insertRightAndRotate(pleft, hash(ileft), hash(id, uint160(0)), rotation) end

As we can see, there are 3 Merkle proofs for insertion (and two for deletion). Insertion or deletion of cells is more unusual procedure than reading or writing values inside the snapp.

Even if we add the cell before each read operation inside the snapp, we have a number of constraints similar to SMT case. For more used operations (for example, transfer of tokens from cell to cell) we have advantage ~3.5 lesser length of Merkle proof.

Authors

snjax

fomalhaut88