A week ago, I posted an SMT construction onA nearly-trivial-on-zero-inputs 32-bytes-long collision-resistant hash function

The advantage of this construction encourage me to post it again:

- this construction does not lose any security compare to others
- no pre-calculated hash set, which is perfect for contracts.

After a week of work, I optimized the storage, and now it takes 2n + log(n)

size to store an SMT; log(n)

times hashing and inserting for key-value updating.

The code is implemented in Rust https://github.com/jjyr/sparse-merkle-tree

Pros:

- no pre-calculated hash set
- support both exist proof and non-exists proof
- efficient storage and key updating

This is just another construction of SMT, no magic improvement compare to other constructions, see this comment:

An optimized compacted sparse merkle tree without pre-calculated hashes[

Data Structure

](/c/data-structure)

The scheme I linked to also doesn't need to do non-trivial hash pre-computation (seethis comment specifically). The scheme I linked to only needs O(\lg n) disk reads (but still requires 256 hashes). In this comment further down however, an optimization is presented that reduces the number of hashes to O(\lg n) (hint: it's an isomorphism of the optimization you propose here, which you'd know if you had bothered to read the prior art).

To save your time, you can stop reading if you already follow the discussions mentioned by the comment.

Trick 1: Optimized hash function for 0 value.

We define the hash merge function as below:

1. if L == 0, return R
2. if R == 0, return L
3. otherwise sha256(L, R)

Follow this rule; an empty SMT is constructed by zeros nodes; it brings an advantage: we do not need a pre-calculated hash set.

This function has one issue, merge(L, 0) == merge(0, L)

, which can easily construct a conflicted merkle root from a different set of leaves.

To fix this, instead of use hash(value)

, we compute a leaf_hash = hash(key, value)

as leaf's value to merge with its sibling.

(we store leaf_hash -> value

in a map to indexed to the original value).

Security proof:

- Since the key is hashing into the leaf_hash, no matter what the value

is, the leaves' hashes are unique.

- Since all leaves have a unique hash, nodes at each height will either merged by two different hashes or merged by a hash with a zero(for a non-zero parent, since the child hash is unique, the parent is surely unique at the height).

- For the root, if the tree is empty, we got zero, or if the tree is not empty, the root must merge from two hashes or a hash with a zero, it's still unique.

So from construction, this SMT is security because we can't get a collision root hash.

Trick 2: A node structure to compress the tree.

The classical node structure is Node {left, right}

, it works fine if we insert every node from root of the tree to bottom, but mostly node is duplicated, we want our tree only store unique nodes.

The idea is simple: for a one leaf SMT, we only store the leaf itself, when inserting new leaves, we magically extract location info from internal tree node, and decide how to merge nodes.

The key to this problem is the "key". Each key in the SMT can be seen as a path from the root of the tree to leaves, so the idea is we store the key in node, and when we need to merge two nodes, we extract the location info from the keys:

We can calculate the common height of two keys, which is exactly the height the two nodes be merged.

```
fn common_height(key1, key2) {
for i in 255..0 {
   if key1.get_bit(i) != key2.get_bit(i) {
      // common height
      return i;
   }
}
return 0;
}
```

The node structure BranchNode { fork_height, key, node, sibling}

, can use one unique node to expression all duplicated nodes on the "key" path between height [node.fork_height, 255]

.

- fork_height

is the height when the node created; for a leaf, it is 0.

- key

is the key of a child's key when constructing the node. for a leaf node, the key is leaf's key.

- node

and sibling

is like the left

and right

in the classical structure; the only difference is their position is not fixed.

To get a left child of a node in height H

:

1. check H

-th bit of key

1. if it is 1

means the node

is on the right at height H

, so sibling

is the left child

1. if the 0

means the node

is on the left

```
// get children at height
// return value is (left, right)
fn children(branch_node, height) {
let is_rhs = branch_node.key.get_bit(height);
if is_rhs {
    return (branch_node.sibling, branch_node.node)
} else {
    return (branch_node.node, branch_node.sibling)
}
}
```

To get a key from SMT, we walk down the tree from root to bottom:

```
fn get(key) {
let node = root;
// path order by height
let path = BTreeMap::new();
loop {
    let branch_node = match map.get(node) {
        Some(b) => b,
        None => break,
    }
    // common height may be lower than node.fork_height
    let height = max(common_height(key, node.key), node.fork_height);
    if height > node.fork_height {
        // node is sibling, end search
        path.push(heignt, node);
        break;
    }
    // node is parent
    // extract children position from branch
    let (left, right) = children(branch_node, height);
    // extract key positon
    let is_right = key.get_bit(height);
    if is_right {
        path.push(height, left);
        node = right;
```

```
    } else {
        path.push(height, right);
        node = left;
    }
}
return self.leaves[node];
}
```

There is nothing special for other operations: updating, merkle proof. It just works as expected.