Set Merkle Tree

The Set Merkle Tree is an authenticated data structure representing a set of `Nullifier`s, primarily supporting:

- Insertion
- Proofs of inclusion/exclusion

Cryptographic primitives

The Set Merkle Tree uses 3 hash functions

- `H_elem   : Nullifier       -> Digest`
- `H_leaf   : Nullifier       -> Digest`
- `H_branch : (Digest,Digest) -> Digest`

Here, `Digest` is a bit-array type of some fixed length `N`, and we assume:

- There is some special value `EMPTY_HASH: Digest`, such that finding a preimage of `EMPTY_HASH` for any `H` in `{H_elem,H_leaf,H_branch}` is infeasible.
- It is infeasible to find any values `(H1,x)`, `(H2,y)` such that `(H1,x) != (H2,y)`, `H1,H2 \in {H_elem,H_leaf,H_branch}`, and `H1(x) == H2(y)`.
  - This implies that `H_elem`, `H_leaf`, and `H_branch` are collision-resistant
  - This also implies that finding any overlap in the images of the hashes is infeasible, eg, finding any values `x` and `y` such that `H_branch(y) == H_elem(x)`.

In the implementation:

- N == 512, since `Blake2B` has a 64-byte output
- `EMPTY_HASH` is the all-zero string.
- `H_elem(nul)` is `h(canonical_serialize(nul))` where `h` is `Blake2B` personalized with "CAPSet Elem"
- `H_leaf(nul)` is `h(canonical_serialize(nul))` where `h` is `Blake2B` personalized with "CAPSet Leaf"
- `H_branch(nul)` is `h("l"||l||"r"||r)` where `h` is `Blake2B` personalized with "CAPSet Branch"

We also assume that any sparse array `arr` of size `2^N` has less than `negl(N)` non-empty elements.

Ideal Trees

Given a set of nullifiers `arr_S = {nul_1,...,nul_k}`, we define the set merkle tree by first converting it into a sparse array `arr_S` of size `2^N`, such that:

- for each `i \in {1,...,k}`, `arr_S[LittleEndian(H_elem(nul_i))] == nul_i`

- for all other indices `ix`, `arr_S[ix] == <EMPTY>`

where LittleEndian converts a little-endian bitstring into a natural number, ie:

```
LittleEndian([]) = 0
LittleEndian(arr) = arr[0] + 2*LittleEndian(arr[1..])
```

Now we can define the "ideal tree" `IT` of an array inductively:

```
- IT([<EMPTY>]) = ITEmptyLeaf
- IT([elem]) = ITLeaf(elem)
- IT(arr) = ITBranch(
    IT(arr[0..(length(arr)/2)]),
    IT(arr[(length(arr/2))..length(arr)]))
```

Note that IT is injective.

The hash of the Set Merkle Tree is then defined on `IT(arr_S)`:

```
- hash(ITEmptyLeaf) = EMPTY_HASH
- hash(ITLeaf(elem)) = H_leaf(elem)
- if hash(l) == EMPTY_HASH and hash(r) == EMPTY_HASH,
    then hash(ITBranch(l,r)) = EMPTY_HASH
    else hash(ITBranch(l,r)) = H_branch(hash(l),hash(r))
```

The actual implementation includes several modifications to make the computational and memory cost lower than directly working with `IT`, but in a correct implementation all operations will match the result on `IT`.

Inserting into an ideal tree is done by:

```
IT_insert(t: IT, nul: Nullifier) -> IT:
    let b_0,...,b_{N-1} = H_elem(nul);
    return IT_insert_inner(t,nul,[b_{N-1},b_{N-2},...,b_0])

IT_insert_inner(t: IT, nul: Nullifier, path: [bit]) -> IT:
    match t {
        ITEmptyLeaf => {
            assert(path == []);
            return ITLeaf(nul);
        }
        ITLeaf(elem) => {
            assert(path == []);
            assert(elem == nul);
            return ITLeaf(elem);
        }
        ITBranch(l,r) => {
            if path[0] == 1 {
                return ITBranch(l,IT_insert_inner(r,nul,path[1..]));
            } else {
                return ITBranch(IT_insert_inner(l,nul,path[1..]),r);
```

```
        }
    }
}
```

Lemma: for any arr,nul,path, such that 2^length(path) == length(arr),

```
    IT_insert_inner(IT(arr),nul,path)
 == IT(arr[LittleEndian(path) := nul])
```

where `arr[ix := y]` is the array `arr2` such that if `i != ix`, `arr2[i] == arr[i]`, and `arr2[ix] == y`.

Proof: by induction.

Theorem: For any arr,nul such that 2^N == length(arr),

```
   IT_insert(IT(arr),nul)
== IT(arr[LittleEndian(H_elem(nul)) := nul])
```

Proof: Apply the lemma.

A proof of inclusion for `elem` in `S` is a sequence of `Digests` `sib_0,...,sib_{N-1}` of "sibling subtrees", ie:

```
sib(i,elem) -> Digest:
    let b_0,...,b_{N-1} = H_elem(elem);
    let sib_lower = LittleEndian([0,0,...0,,(1-b_i),b_{i+1},...,b_{N-1}]);
    let sib_upper = LittleEndian([1,1,...1,,(1-b_i),b_{i+1},...,b_{N-1}]);

    return hash(IT(arr_S[sib_lower..(sib_upper+1)]));
```

Conceptually, if we walk down the ideal tree to where `elem` would get inserted, we will hit many `ITBranch(l,r)` nodes. If we write down the hashes of the subtrees we don't go into (ie, if insertion goes into r, write down `hash(l)`, and vice versa), this will be `sib_{N-1},...,sib_0`.

A proof of exclusion for `elem` is similar, except it is `sib_i,...,sib_{N-1}` for some `i` such that following the path `b_{N-1},...,b_i` brings us to an empty subtree.

Checking each type of proof amounts to calculating `hash(IT(arr_S))` using these sibling hashes, and checking that it matches.

Theorem: inclusion and exclusion proofs are complete. Proof: Follows from the definition of `hash(IT(arr_S))`.

Theorem: Given an honestly constructed root hash `H = hash(IT(arr_S))`, inclusion and exclusion proofs are computationally sound. Proof: Suppose we have an adversary that can generate a false inclusion or exclusion proof for `elem`. Let `b_0,...,b_{N-1} = H_elem(elem)`.

This means there are two sequences

```
g_i,g_{i+1},...,g_N
h_j,h_{j+1},...,h_N
```

such that:

- for all k in {i,. . . ,N-1}, if b__k == 0, g__{k+1} == hash(ITBranch(g__k,sib__k)) if b__k == 1, g__{k+1} == hash(ITBranch(sib__k,g__k))
- for all k in {j,. . . ,N-1},, if b__k == 0, h__{k+1} == hash(ITBranch(h__k,sib__k)) if b__k == 1, h__{k+1} == hash(ITBranch(sib__k,h__k))
- g__N == h__N == H
- if `elem` is in `S`, `i == 0`, `g_i = H_leaf(elem)`, and `h_j == EMPTY_HASH`
- if `elem` is not in `S`, `j == 0`, `g_j = H_leaf(elem)`, and `g_i == EMPTY_HASH`

From these two sequences, we can walk "down the tree" from `N` to `0`, and find one of the following:

(a) some `k` such that `g_{k+1} == h_{k+1}` but `g_k != h_k`, which yields a collision of `H_branch`.
(b) `j > 0` and `g_j == h_j == EMPTY_HASH` or `i > 0`, and `g_i == h_i == EMPTY_HASH` which yield a preimage of `EMPTY_HASH` for `H_branch`.
(c) `g_0 == h_0`, which yields a preimage of `EMPTY_HASH` for `H_leaf`

each of which violates our cryptographic assumptions.

Corollary: For any `t1: IT(arr_S1)`, `t2: IT(arr_S2)`, if `hash(t1) == hash(t2)`, then under our computational assumptions, `arr_S1 == arr_S2`.

Proof: Let `n1` be the number of non-empty slots in `arr_S1` and `n2` the same for `arr_S2`. By assumption, `n1 < negl(N)` and `n2 < negl(N)`. If there is any slot which differs between `arr_S1` and `arr_S2`, we can calculate a proof of inclusion relative to the other, violating computational soundness.

NOTE: because we're assuming that collisions are infeasible, N needs to be large enough such that `|S| < negl(N)`, since at ~2^(N/2) you will start encountering birthday-bound collisions.

Practical Trees

The ideal tree decription can be optimized to have a smaller memory footprint and smaller proofs. Let's call this the "Practical Tree" (PT). For the same array, there may be multiple valid PTs, so we will define PTs by an inductive predicate instead of as a function:

```
PTEmptySubtree is PT(height,arr) if:
    arr == [<EMPTY>,<EMPTY>,...,<EMPTY>], and
    length(arr) == 2^height
PTLeaf(height,elem) is PT(height,arr) if:
    arr == [<EMPTY>,...,<EMPTY>,elem,<EMPTY>,...,<EMPTY>] (ie, arr only
        contains `elem`), and
```

```
    length(arr) == 2^height
PTBranch(l,r) is PT(height,arr) if:
    length(arr) == 2^height, and
    l is not PTEmptySubtree and
    r is not PTEmptySubtree and
    l is PT(height-1,arr[0..(length(arr)/2)]), and
    r is PT(height-1,arr[(length(arr)/2)..length(arr)]))
```

For simplicity, PT forbids branch nodes from having two empty children.

If `arr` is well-formed (ie, there is some set S such that arr == arr_S), a PT can be "idealized" by:

```
PT_idealize(height, t: PT(height,arr)) -> IT:
    if t is PTEmptySubtree:
        if height == 0:
            return ITEmptyLeaf
        else:
            return ITBranch(PT_idealize(height-1, PTEmptySubtree),
                            PT_idealize(height-1, PTEmptySubtree))
    else if t is PTLeaf(height,elem):
        if height == 0:
            return ITLeaf(elem);
        else:
            let B = [b_0,...,b_{N-1}] = H_elem(elem);
            if B[height-1] == 1:
                return PT_idealize(height,
                                   PTBranch(PTEmptySubtree,
                                            PTLeaf(height-1,elem)))
            else:
                return PT_idealize(height,
                                   PTBranch(PTLeaf(height-1,elem),
                                            PTEmptySubtree))
    else t is PTBranch(l,r):
        return ITBranch(PT_idealize(height-1,l),PT_idealize(height-1,r));
```

Theorem: if `S` is a nullifier set, `arr_S` is its array representation, `length(arr_S) == 2^height`, and `pt` is `PT(height,arr_S)`, then `PT_idealize(height, pt) == IT(arr_S)`

Proof: Induction over pt. The tricky case is `PTLeaf(height,elem)` with `height > 0`. In this case, `pt` is representing an array `a_pt = [<EMPTY>,...,<EMPTY>,elem,<EMPTY>,...,<EMPTY]`. `a_pt` is a subarray of `arr_S`, whose bounds are determined by the most significant bits of `H_elem(elem)`. Thus, the index of `elem` in `a_pt` is `a_pt_ix = LittleEndian(H_elem(elem)[..height])`, ie, the number represented by the `height` least significant bits of `H_elem(elem)`. So when we convert `PTLeaf(height,elem)` into a branch, we need to figure out

which side of the array elem is on, so we read the most significant of the lowest `height` bits of `H_elem`, which tells us if `a_pt_ix` is in `{0,...,(2^(height-1) - 1)}` or `{2^(height-1),...,(2^height - 1)}`, and then we put `PTLeaf(height-1,elem)` on the appropriate side of a branch and continue idealizing.

Hashing the tree is done by copying the ideal tree pattern:

```
    - hash(PTEmptySubtree) = EMPTY_HASH
- if H_elem(elem)[height-1] == 1,
        then hash(PTLeaf(height,elem))
            = H_branch(EMPTY_HASH,hash(PTLeaf(height-1,elem)))
        else hash(PTLeaf(height,elem))
            = H_branch(hash(PTLeaf(height-1,elem)),EMPTY_HASH)
- hash(PTBranch(l,r)) = H_branch(hash(l),hash(r))
```

By induction, if `pt: PT(height,arr_S)`, then `hash(pt) == hash(PT_idealize(pt))`.

Proofs of inclusion/exclusion now can either be a path to an empty subtree, or a path to a singleton subtree. If the singleton subtree is `PTLeaf(height,leaf_elem)` and `leaf_elem == elem`, it is a proof of inclusion. Otherwise, it's a proof of exclusion. Since PT hashing matches ideal tree hashing, these proofs are also complete and computationally sound.

Sparse Representation

One last useful construction is included in our implementation: a sparse in-memory representation of PTs.

If there is some PT `pt`, and you just want to be able to check proofs of inclusion/exclusion, you only need to keep hash(pt) around. Since the proof system is computationally sound, you know that any proof that succeeds is respresentative of the underlying set. However, one can think of an inclusion/exclusion proof not just as a proof that a particular element is included/excluded, but also as a "reminder" of the parts of the tree which are relevant to the query "is `elem` in S?". In fact, because the proof gives you all the necessary sibling hashes, a proof of non-inclusion of `elem` can be used to calculate `hash(PT_insert(pt,elem))`. It also allows you to update other proofs – if I have proofs that `x` and `y` are not in `S`, I can calculate proofs that `x` is in `insert(S,x)` and that `y` is not. The mechanisms of "remember" and "forget" let us generalize this strategy.

First, let's define our "forgetful tree", `FT(height,arr)`:

```
FTEmptySubtree is FT(height,arr) if:
    arr == [<EMPTY>,<EMPTY>,...,<EMPTY>], and
    length(arr) == 2^height
FTLeaf(height,elem) is FT(height,arr) if:
    arr == [<EMPTY>,...,<EMPTY>,elem,...,<EMPTY>] (ie, arr only
        contains `elem`), and
```

```
    length(arr) == 2^height
FTBranch(l,r) is FT(height,arr) if:
    length(arr) == 2^height, and
    l is not FTEmptySubtree and
    r is not FTEmptySubtree and
    l is FT(height-1,arr[0..(length(arr)/2)]), and
    r is FT(height-1,arr[(length(arr)/2)..length(arr)]))
FTForgottenSubtree(h) is FT(height,arr) if:
    there exists some pt: PT(height,arr) such that h == hash(pt)
```

Any `pt: PT(height,arr)` can be mapped canonically to some `ft: FT(height,arr)` by mapping `PTEmptySubtree`, `PTLeaf`, and `PTBranch` to `FTEmptySubtree`, `FTLeaf`, and `FTBranch` respectively.

Hashing FTs is the same as with PTs, except that:

- `hash(FTForgottenSubtree(h)) = h`

Insertion into an FT can fail if the value would go into a `FTForgottenSubtree`. However, if it does not, then insertion behaves exactly identically to insertion into a PT.

We'll call `ft: FT(height,arr_S)` "well-formed" if there is some `pt: PT(height,arr_S)` such that `hash(ft) == hash(pt)`.

Mapping a `pt: PT(height,arr_S)` to its canonical FT clearly creates a well-formed tree.

Inserting into a well-formed FT also preserves a well-formed FT, since insertion only succeeds when it would exactly mimic the PT operation.

The last 2 operations are `FT_remember(ft, elem, proof)` and `FT_forget(ft, elem)`:

```
FT_forget(t: FT(height,arr_S), nul: Nullifier)
        -> (FT(height,arr_S), is_present: bool, presence_proof):
    let b_0,...,b_{N-1} = H_elem(nul);
    return FT_forget_inner(t,nul,[b_{N-1},b_{N-2},...,b_0])

FT_forget_inner(t: FT(height,arr_S), nul: Nullifier, path: [bit])
        -> (FT(height,arr_S), is_present: {true,false,unknown}, presence_proof):
    match t {
        FTEmptySubtree => {
            assert(path == []);
            return (FTEmptySubtree,false,[]);
        }
        FTLeaf(height,elem) => {
            assert(path == []);
            if elem == nul {
                return (FTForgottenSubtree(hash(t)),
```

```
                        true,
                        [FTLeaf(height,elem)])
            } else {
                return (t,false,[FTLeaf(height,elem)]);
            }
        }
        FTBranch(l,r) => {
            if path[0] == 1 {
                let (new_r,is_present,proof) =
                    FT_forget_inner(r,nul,path[1..]);
                return (ITBranch(l,new_r),
                        is_present,
                        proof + [hash(l)]);
            } else {
                let (new_l,is_present,proof) =
                    FT_forget_inner(l,nul,path[1..]);
                return (ITBranch(r,new_l),
                        is_present,
                        proof + [hash(r)]);
            }
        }
        FTForgottenSubtree(h) => {
            return (t, unknown, []);
        }
    }

FT_remember(t: FT(height,arr_S), nul: Nullifier, proof)
        -> FT(height,arr_S):
    Check (proof,nul) relative to hash(t);
    let b_0,...,b_{N-1} = H_elem(nul);
    return FT_remember_inner(t,nul,proof,[b_{N-1},b_{N-2},...,b_0])

FT_remember_inner(t: FT(height,arr_S), nul: Nullifier, proof, path: [bit])
        -> FT(height,arr_S):
    match t {
        FTEmptySubtree => {
            assert(false);
        }
        FTLeaf(height,elem) => {
            // in this case, `proof` must be a non-inclusion
            // proof. No action is necessary.
            return t;
        }
        FTBranch(l,r) => {
            if path[0] == 1 {
                let new_r = FT_remember_inner(
```

```
                r, nul, proof[..length(proof)-1], path[1..]
                );
            return ITBranch(l,new_r);
        } else {
            let new_l = FT_remember_inner(
                l, nul, proof[..length(proof)-1], path[1..]
                );
            return ITBranch(new_l,r);
        }
    }
    FTForgottenSubtree(h) => {
        if length(proof) == 1 {
            // proof[0] is either FTLeaf(height,elem) or
            // FTEmptySubtree
            return proof[0];
        } else {
            let sib_hash = path[length(path)-1];
            // the exact hash value we use here isn't
            // used, so default it to 0.
            let my_subtree = FT_remember_inner(
                FTForgottenSubtree(0), nul,
                proof[..length(proof)-1], path[1..]);
            if path[0] == 1 {
                return FTBranch(FTForgottenSubtree(sib_hash),
                                my_subtree);
            } else {
                return FTBranch(my_subtree,
                                FTForgottenSubtree(sib_hash));
            }
        }
    }
}
```

The last correctness property is that:

- `forget`/`remember` preserve well-formedness
- `remember` will always succeed given a correct proof
- `forget` yields correct proofs
- `forget` and `remember` do not change the hash of the tree.
- `forget` removes the relevant path from the "in-memory" tree, and `remember` restores it