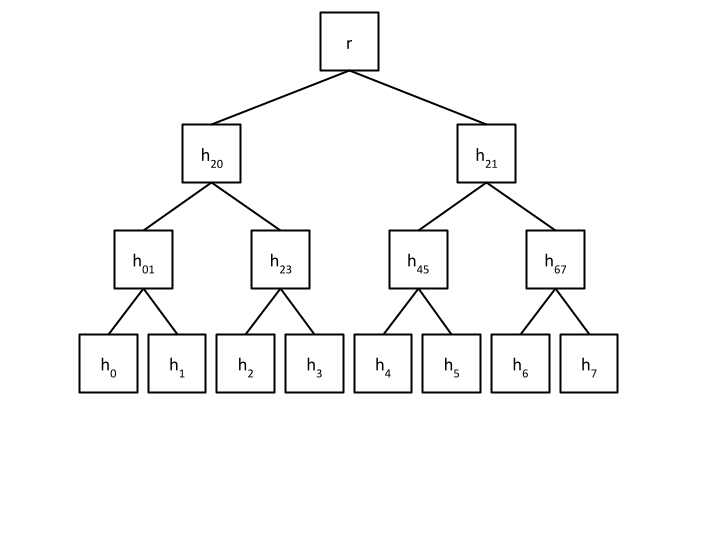
**Question 1: Merkle Trees, Mixers, & Tornados, oh my!**

Here is a Merkle Tree:



1. Let the hash of some data d3 = h3. What is the minimal set of hashes you need to provide to prove d3 is in a block.

Answer: We need to provide

2. Write an algorithm in Solidity pseudocode for doing this. (Assume all data hashes are provided in an array and Merkle Tree stores 8 transactions).

function verify(arrayOfHashes, root, hash, index)

/\* arrayOfHashes -> array of hashes that needed to compute

\* a merkle root

\* root -> merkle root itself

\* hash -> hash we want to proof (d3 in our example)

\* index -> start index of a hash (4 in our example)

\* hashFunction is a keccak256 function in solidity that will

\* combine our hashes

\*/

for (i = 0; i < arrayOfHashes.length; i++) {

proofElement = arrayOfHashes[i]

if (index mod 2 == 0) {

hash = hashFunction(hash, proofElement)

} else {

hash = hashFunction(proofElement, hash)

}

index = index div 2

// div is an integer division (3 div 2 = 1)

}

return bool(hash == root)

}

}

3. **Implement your pseudocode in Solidity.**

contract MerkleProof {

function verify(

bytes32[] memory arrayOfHashes,

bytes32 root,

bytes32 leaf,

uint index

) public pure returns (bool) {

bytes32 hash = leaf;

for (uint i = 0; i < arrayOfHashes.length; i++) {

bytes32 proofElement = arrayOfHashes [i];

if (index % 2 == 0) {

hash = keccak256(abi.encodePacked(hash, proofElement));

} else {

hash = keccak256(abi.encodePacked(proofElement, hash));

}

index = index / 2;

}

return hash == root;

}

}

4. In an interoperable bridge:

Explain why the chain identifier is necessary to prevent double-spending.

Sketch the ZKP needed to check this.

While creating big network of merkle roots, it can be a problem that a user withdraws some funds from one network (like eth) and then used the same nullifier to withdraw from other network (bsc). So we want to add chain identifier to our root system like R = [R\_1, R\_2,…]

And R\_1=[i,s\_1,s\_2,…] I – chain identifier

5. In a mixer, prove how to prevent double spending with another scheme beyond hashing the first element of the secret set.

We choose the hash of one element of the preimage and expose that as a public

input which we call the **nullifier**

Nullifier is also a proof of knowledge nullifier = H(s\_1,s\_1).

So, we do that to prevent user from double-spending. If a user already used that nullifier, C and R have changed, user can’t use this nullifier anymore.

6. Describe the variables and constraints needed to check if the hash of a nullifier and a random number is in a list of hash values without revealing what the hash is. Describe why we may want to do this.

We require one of the public inputs to be tied directly to the preimage of C, where C is a proof of knowledge and it is in a merkle tree with a root R.

[C=H(s\_1,s\_2),…] (s\_1,s\_2 are secret values)

Nullifier is also a proof of knowledge nullifier = H(s\_1,s\_1).

So, we do that to prevent user from double-spending. If a user already used that nullifier, C and R have changed, user can’t use this nullifier anymore. We don’t want to reveal a hash of nullifier because we don’t want anyone use our nullifier.