Introduction to Merkle Trees

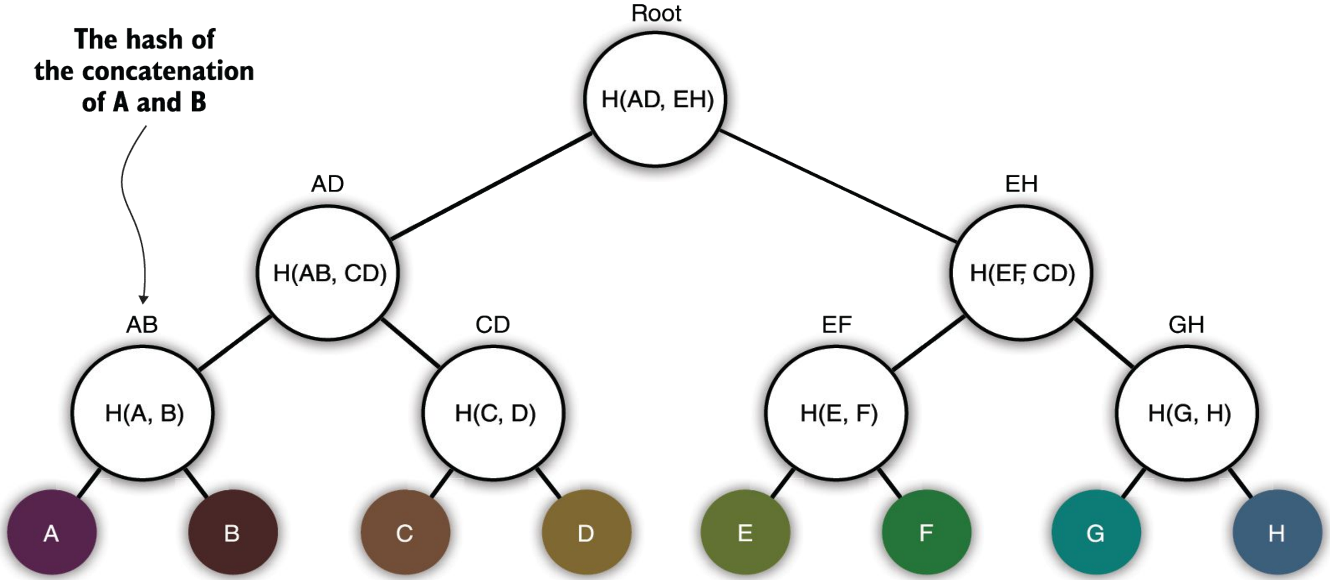
A brief overview of Merkle trees, use cases, and security properties

What is a Merkle tree?

- A Merkle tree is a **hash tree** structure that is used to efficiently verify the integrity of a set of data items.
- Named after its creator, **Ralph Merkle**, who first described it in 1979.
- Merkle trees are a useful data structure that is used in a variety of real-world applications and protocols.
- Merkle trees are a fundamental part of blockchain consensus, BitTorrent data integrity, IPFS, and other distributed systems that require efficient and secure data verification.
- Merkle trees have many variants, including sparse Merkle trees, Merkle Patricia trees, and Verkle Trees.

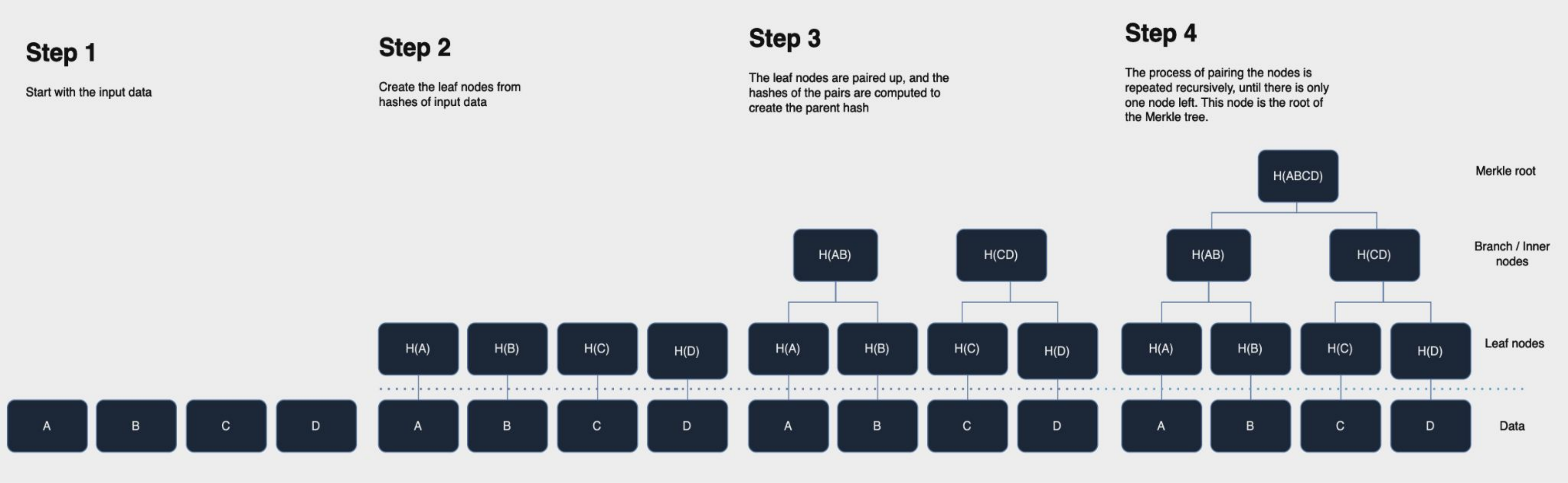
What is Merkle Tree?

- A Merkle tree is a **binary tree** data structure, where each parent node contains the hash of it’s children. The **root** node of the tree contains the hash of all of the data items in the set.
- A **verifier** only need a subset of the tree (data set) to validate membership inclusion (e.g. light client).
- A **prover** who have access to the entire tree is able to generate a proof for a given leaf node.



Merkle Tree Construction

- Merkle trees are binary trees, constructed recursively from the bottom up.
- The first step is to hash each data item in the set.
- The next step is to pair up the hashes and hash the pairs.
- This process is repeated until there is only one hash left, this hash is the root hash of the Merkle tree.



Step 1

Start with the input data

Step 2

Create the leaf nodes from hashes of input data

Step 3

The leaf nodes are paired up, and the hashes of the pairs are computed to create the parent hash

Step 4

The process of pairing the nodes is repeated recursively, until there is only one node left. This node is the root of the Merkle tree.

Membership Proof

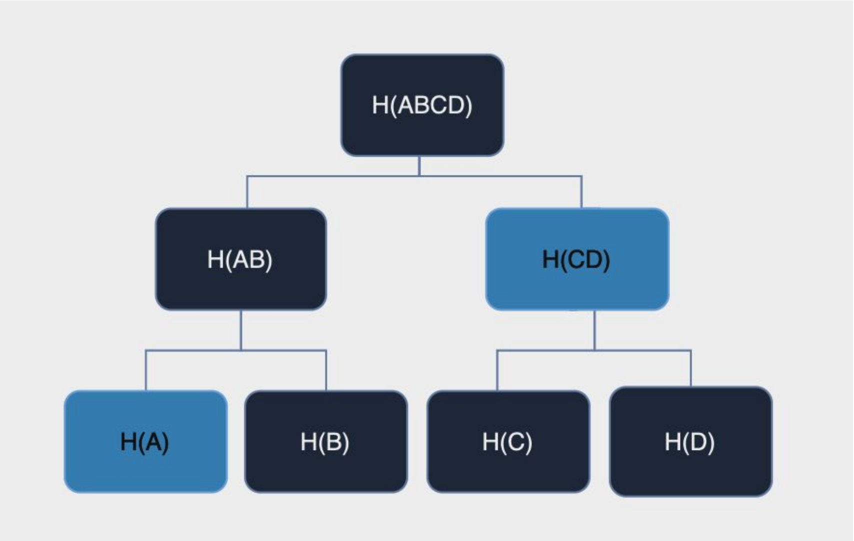
A **Merkle Proof** is a set of hashes that can be used to prove a given leaf's (hash) membership in the tree.

To prove that a specific leaf node is in a Merkle tree, a **verifier** can calculate a minimal path from the leaf node to the root of the tree, following the sibling nodes. The verifier can then calculate the Merkle root of the path and compare it to a known (trusted) root hash.

Membership Proof (Example)

To generate a proof that proves membership of A, we take following steps to generate the proof:

1. Start with the leaf node A and identify its sibling node B (denoted as H(B) where H represents the hash function).
2. Traverse the tree to the next level up. The sibling at this level is the hash of node CD, denoted as H(CD).
3. The final proof path is H(B), H(CD), derived from the sibling nodes along the path from the target node A to the root.



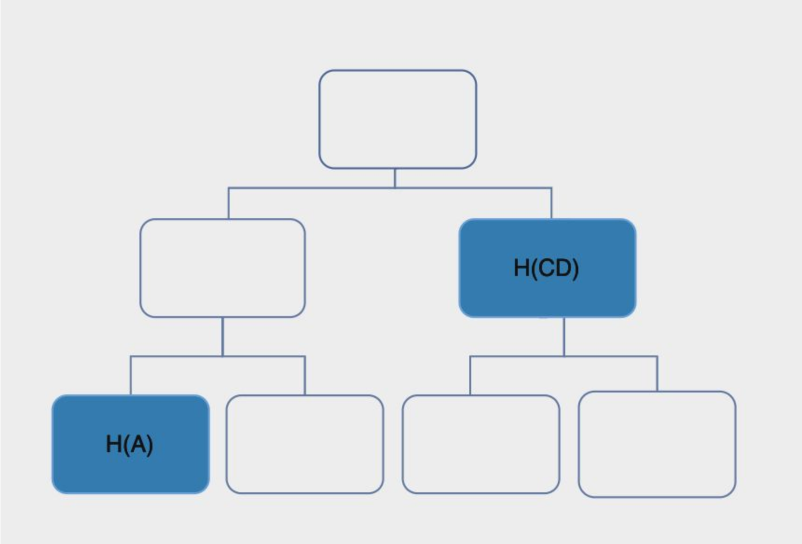
To prove A membership, Proof A = H(B), H(CB)

Membership Proof (Example)

The **verifier** needs the smallest amount of data - the root, the specific node of interest, and all sibling nodes along the path to the root. A Merkle Tree with n leaves has **O(log2 n)** sized proofs. In large tree

Given node A, to generate the proof we compute the hash H(AB), and then hash it with H(CD) to get H(ABCD).

To verify the membership, compare this computed hash (H(ABCD)) with the known (trusted) root hash of the Merkle Tree. If they match, it proves that A is indeed part of the tree.



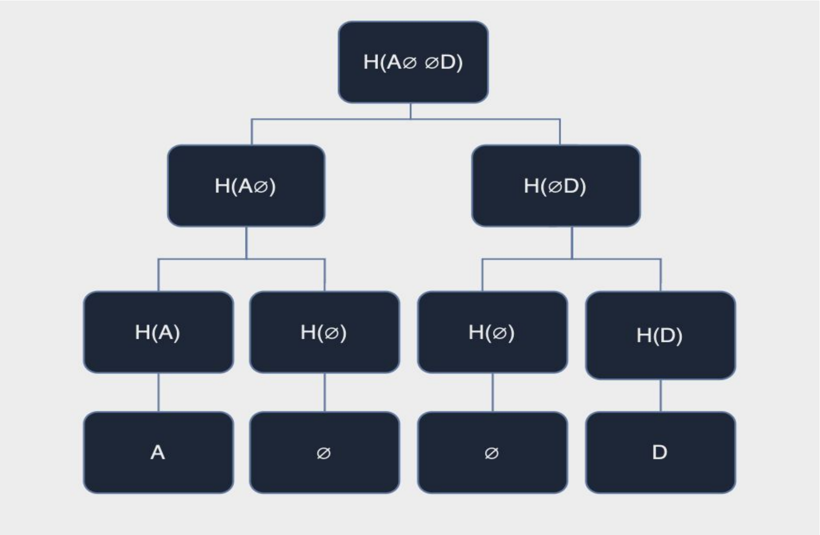
To prove A membership, with proof = H(B), H(CB).

Variants of Merkle Trees

- **Sparse Merkle trees**
  - Like standard Merkle Tree, except data is indexed. This means each data point is placed at the leaf corresponds to it's index.
- **Merkle mountain Rang**e
  - It is a list of of Merkle trees, each Merkle tree in the list represents a range of data. This merkle tree can be non-balanced
- **Trie**
  - A trie (for re**trie**val), also called digital tree or prefix tree is a type of k-ary search tree, a tree data structure used for locating specific keys from within a set
- **Merkle Patricia Trie**
  - Merkle Patricia trees are a combination of Merkle trees and tries.
  - Standard MPT used in Ethereum with 16 bit RLP prefix
- **Verkle trees**
  - Verkle trees are a new cryptographic construction that was first introduced by John Kuszmaul in 2018.
  - K-ary tree, can have many child nodes
  - Uses vector commitments to reduce the size of proofs and proof generation time.

Sparse Merkle Tree

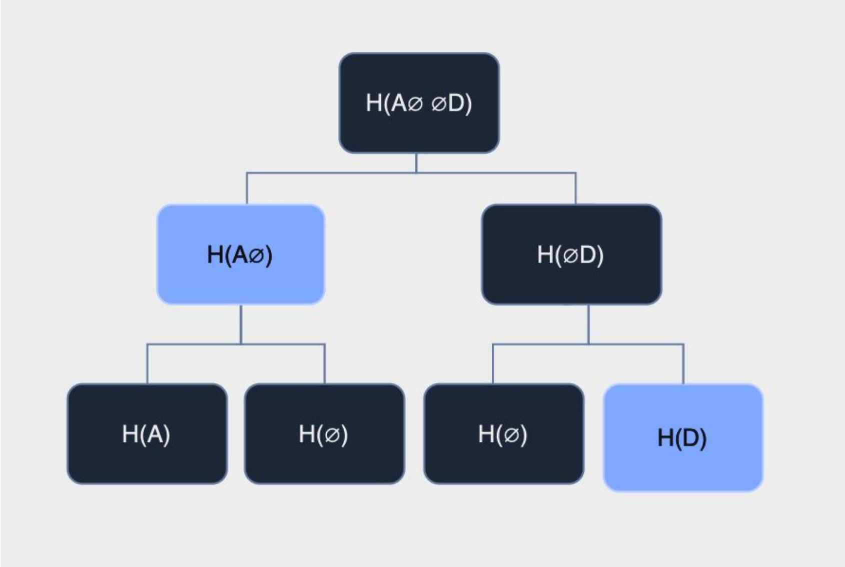
- A Sparse Merkle Tree (SMT) is a data structure that stores data points **indexed** by a unique index.
- The index is typically derived from the hash of the key.
- Spars: the vast majority of the leaves in the tree are **empty**, since most potential indices will not have a corresponding key-value pair.
- Can generate **non-membership proof**
- SMTs allow for efficient updates and lookups, making them an important data structure in distributed systems, including blockchain scaling like Ethereum **Plasma** chains.



Sparse Merkle Tree (Continued)

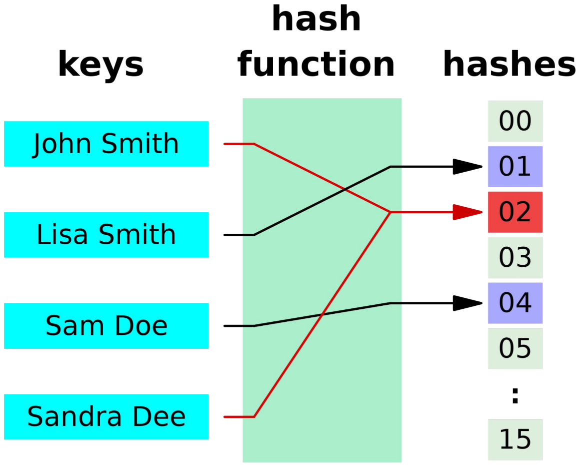
Membership Proof: Works similar to standard merkle tree

Non-Membership Proof: To prove non-inclusion for "C", we know that "C" would be the 3rd leaf node from the left, if it was part of the tree, and so a standard merkle proof would simply return 'null'.



Hash Functions

- Hash functions take data of **arbitrary** size and produce a **fixed-size** output. This means that a hash function can be used to create a unique identifier for any piece of data, regardless of its size.
- ***Deterministic***: always produces the same result if given the same input



Security Properties of Hash Functions

- Pre-image resistance
  - One-wayness, no one should be able to reverse the function.
  - ***Caveat***: You can’t hide something that’s too small or predictable.
- Second pre-image resistance
  - An attacker with a given fixed input, shouldn’t be able to second different input which produces same hash or digest.
- Collision Resistance:
  - An attacker shouldn’t be able to find any two different inputs which produces the same digest. This is different from the previous property where an attacker is given a fixed input (m1), and tries to find second input (m2) which produces the same digest.

Security Considerations of Hash Functions

- The security properties of hash functions are **meaningless** on their own.
- Random oracle test: If the input is predictable, then it is possible to find collisions or preimages, even if the hash function is secure.
- **The size of the digest matters**. A hash function with a small output size is more susceptible to collisions. In practice, a hash function should have a minimum output size of 256 bits.
- Truncating the digest can reduce security. If the digest is truncated, then it is possible to find collisions or preimages more easily. In order to achieve 128-bit security at a minimum, a digest must not be truncated under:
- Hash functions are **not perfect**. It is possible to find collisions or preimages for even the most secure hash functions. However, the probability of finding a collision or preimage is very small, and it would take an enormous amount of computing power to do so.
- Hash functions are **not a replacement for encryption**. Hash functions are designed for **integrity** not for confidentiality.

Attacks in Merkle Trees

- Merke trees as a data structure are secure by design.
- Since they’re hash trees they inherit the same security properties as their underlying hash function.
- It’s possible to reproduce MerkleProof from intermediate nodes.
- Poor implementation can make Merkle Tree more susceptible to attacks irrespective of the underlying hash function.
- Data Availability attacks are possible in certain blockchains where light nodes do not download the entire blockchain ledger.

Conclusion

In this presentation, we have covered an introduction and overview of Merkle trees, including some other types like Sparse Merkle Trees. We have also demonstrated an example of how Merkle trees can be used to generate and prove membership. Merkle trees are a powerful tool that can be used to verify the integrity of data. They are used in a variety of applications, including blockchain technology, file verification, and data integrity.

Finally, we have briefly discussed hash functions and their security properties. Hash functions are a critical part of Merkle trees. They provide the foundation for the security of Merkle tree

References

- Real World Cryptography by David Wang: https://www.oreilly.com/library/view/real-world-cryptography/9781617296710/
- Efficient Sparse Merkle Tree: https://eprint.iacr.org/2016/683.pdf
- Verkle tree: https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf
- The Right Way to Hash a Merkle Tree: https://www.youtube.com/watch?v=NfK4np15E64&t=1272s
- Data Availability Attacks in Blockchain: https://arxiv.org/abs/2108.13332
- Merkle Tree Implementation in Golang: https://github.com/mohamedelshami/go-merkle-tree