Merkle Trees

Mahmoud Attia
KAUST
mahmoud.attia@kaust.edu.sa

Abstract

This project begins by describing the structure and operation of Merkle Trees before going on to explain its role in signature amortization in the context of a few of its applications. I will then explain the implementation of a Merkle Tree, followed by a security analysis and performance evaluation.

1 Introduction

1.1 Digital Signatures

Digital signatures are a cryptographic technique used to ensure the authenticity, integrity, and non-repudiation of digital messages. Digital signatures are based upon the Public Key Encryption (PKE) system first proposed by Diffe and Hellman [1].

The process begins with the creation of a pair of keys using a cryptographic algorithm. These keys are private keys, known only to the user and used to create the digital signature, and public keys, which are shared with anyone who wants to verify the signer's digital signature. When a signer wants to sign a document or message, a hash (a fixed-size string of bytes that uniquely represents the data) of the message is created using a hash function. The signer then encrypts this hash with their private key, creating the digital signature. The original message, along with its digital signature, is then sent to the receiver.

To verify a digital signature, the receiver first uses the same hash function to generate a hash of the original message. Then, they decrypt the signature using the sender's public key, obtaining the hash value that the sender encrypted. The signature is verified if the decrypted hash matches the hash generated from the received message [5].

1.2 Signature Amortization

The biggest challenge in using digital signatures for authentication is the computationally intensive nature of the

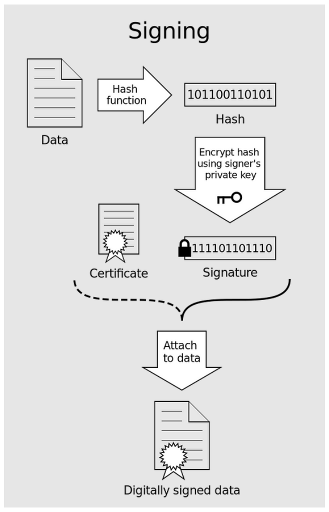

Figure 1: Creating a digital signature [6]

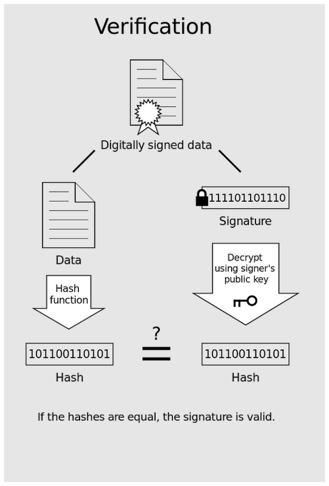

Figure 2: Verifying a digital signature [6]

asymmetric-key-based signatures. In general, making the signature scheme faster comes at the cost of increased space overhead [3].

Amortization is a term commonly used in finance and accounting to describe the process of gradually paying off a debt over time through scheduled, usually equal, payments or to spread the cost of an intangible asset over its useful life. The concept can be applied in different contexts, but the core idea revolves around the gradual reduction of a value or cost over a specified period.

Signature amortization in the context of cryptographic systems refers to techniques used to reduce the computational and storage overhead associated with digital signatures. The idea behind signature amortization is to efficiently manage and minimize these costs by combining multiple signatures into a single one or by creating a system in which a single signature can validate multiple transactions or documents at once. Merkle Trees allow for many pieces of data to be combined and represented by a single hash (a fingerprint of the data). A Merkle tree can be used to create a single signature that authenticates a large number of transactions by signing the root hash of the tree.

By reducing the number of signatures that need to be individually processed, systems can handle more transactions or documents with lower computational costs, making cryptographic systems more scalable and efficient. This concept is particularly relevant in distributed systems, like blockchains, where the efficiency of verifying transactions directly impacts the overall system performance.

1.3 Merkle Trees

Merkle Trees, named after their inventor Ralph Merkle [4], also known as hash trees, are a fundamental data structure in cryptography and computer science, designed to efficiently and securely verify the content of large data structures.

1.3.1 Structure

A Merkle Tree is a binary tree, where each leaf node represents a hash of a block of data, typically a single transaction or a file fragment. Non-leaf (intermediate) nodes are hashes of their respective child nodes.

The structure of a Merkle Tree, as seen in Figure 3, is made up of the following:

- Leaf Nodes: These are at the bottom of the tree and contain the hash of data blocks (data signatures).
- Intermediate Nodes: Each node above the leaf level contains the cryptographic hash of the combined hash values of its two child nodes.
- Root Node (Merkle Root): The single node at the top of the tree, which represents the hash of all the underlying data, summarizing the entire set of data covered by the Merkle Tree.

1.3.2 Creation of a Merkle Tree

To create a Merkle Tree, start by calculating the hash of each piece of data to form the leaf nodes. Then, pair each set of nodes together and hash their combined values to form the nodes of the next layer. Repeat this process layer by layer until you reach a single node at the top, the Merkle Root.

1.3.3 Function of a Merkle Tree

The verifier requests the hash of the specific leaf node along with the hashes of a set of additional nodes necessary to reconstruct the path from the leaf node to the root. This set of nodes is called the Merkle Proof. A proof is comprised of the leaf's sibling, and each non-leaf hash that could not otherwise be calculated without additional leaf nodes. The verifier then computes the hashes step-by-step up the tree by hashing the data block and the provided hashes. If the computed root hash matches the known good Merkle Root, the data block is confirmed to be intact and unaltered [2].

1.4 Applications of Merkle Trees

Merkle Trees are used in real-world applications where efficient and lightweight digital signature verification is paramount. We will explore two notable applications of Merkle Trees.

1.4.1 Blockchain Technology

Blockchain technology prominently uses Merkle Trees to ensure the integrity of transactions within each block of the chain. In a typical blockchain, like Bitcoin:

Each transaction within a block is hashed, and these hashes serve as the leaf nodes of the Merkle Tree. The transactions are then paired and hashed together, and this process is repeated up the tree until a single hash (the Merkle Root) is obtained. This Merkle Root is included in the block's header. The block header also contains the hash of the previous block's header, chaining the blocks securely.

Merkle Trees allow network nodes to verify the presence of specific transactions in a block without needing the entire block's data. They only require the hashes of the branch leading to the transaction. This method is particularly efficient for light nodes in the network, which do not store the entire blockchain.

Any change in a transaction will alter its hash and, subsequently, all hashes up to the Merkle Root, effectively detecting tampering. The blockchain's immutable structure, combined with the Merkle Root in each block, ensures data integrity.

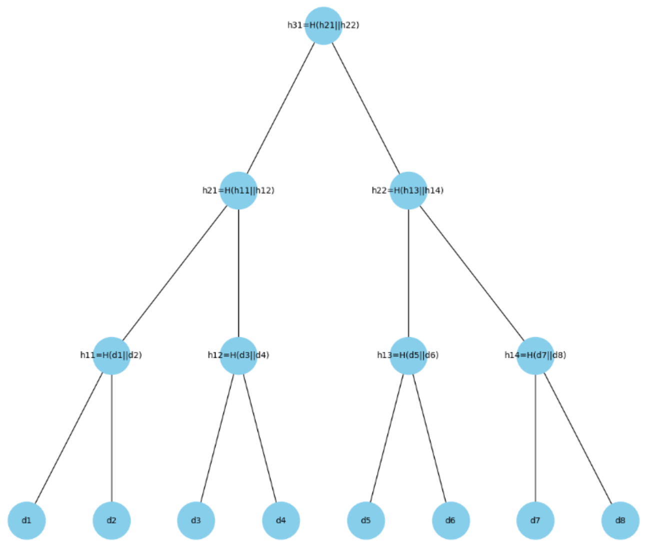

Figure 3: Merkle Tree Structure [2]

1.4.2 Secure Software Distribution

Software distribution platforms can utilize signature amortization to handle the large volume of updates and packages that need to be authenticated before they are delivered to users. A practical implementation is seen in the update mechanisms of large-scale software systems, such as operating system updates, where each update package is signed not individually but rather as a batch.

By using signature amortization, the system only needs to verify a single signature for a batch of updates rather than verifying each update separately. This significantly reduces the computational overhead and speeds up the verification process for users installing multiple updates simultaneously.

Signature amortization ensures that if the batch signature is valid, all individual packages within the batch are untampered and authentic.

2 Implementation

2.1 Tree Construction

We first start by defining a global function, *hash_data* in appendix 7.1, that hashes the input data using the SHA-256 algorithm. This function takes a single input argument, ‘data‘, which can either be a string or bytes. If the input is a string, it will be encoded into bytes using UTF-8 encoding. The function then computes the SHA-256 hash of these bytes and returns the hexadecimal representation of the hash.

We then define the *MerkleTree* class in appendix 7.2. When it first gets initialized, it has 3 initial attributes:

- leaves (list): A list of hashed leaves (data blocks).
- tree (list): A structured list containing all levels of the tree, including the root.
- root (str or None): The root hash of the Merkle Tree. It is None until the tree is computed.

The method, *add_data*, adds data blocks to the leaves list of the Merkle Tree. Each data block is individually hashed and added as a leaf.

After adding the data blocks, we proceed to build the tree from the current leaves using the build_tree function. It starts from the leaves and constructs the tree layer by layer up to the root. Each node in the tree is created by concatenating the hashes of two child nodes and then hashing the concatenated string. The Merkle Tree structure is stored in ‘self.tree‘, where each level of the tree is a list of hashes, starting from the leaves up to the root.

2.2 Proof Generation

The function *get_proof* in appendix 7.3, provides the sequence of hashes (Merkle Path) necessary to verify that a given data block is part of the tree. It returns a list of tuples, each containing a hash and a direction (’L’ or ’R’) indicating whether to append this hash to the left or right of the current hash during verification.

2.3 Proof Verification

The function *verify_proof*, in appendix 7.4, checks whether the provided proof correctly leads from the given data block to the specified Merkle root. Given the list of tuples, each containing a hash and a direction (’L’ or ’R’), it can work up to the root of the tree. If the position is ’L’, it will concatenate the node hash to the left of the current hash and re-hash. Likewise, if the position is ’R’, it will concatenate the node hash to the right of the current hash and re-hash. It then compares the final hash computed from the proof to the provided root.

2.4 Signature Amortization

Within the *MerkleTree* class, the Merkle root is signed using a private key. It signs the Merkle root hash with the private key using the RSA-PSS (Probabilistic Signature Scheme) signature scheme, which is more secure against chosen plaintext attacks compared to the older PKCS#1 v1.5 scheme. The hash function used for both the message digest and the mask generation function (MGF) in PSS is SHA-256. This method is used to authenticate the integrity and origin of the Merkle root, especially in scenarios where the tree structure needs to be verified independently by a third party.

To verify the signature of the Merkle root using a public key, the *verify_signature* method is used, also in appendix 7.5. This method checks if the provided digital signature of the Merkle root can be authenticated with the given public key, ensuring the integrity and authenticity of the Merkle root. It uses RSA-PSS (Probabilistic Signature Scheme) for signature verification, which complements the RSA-PSS scheme used in signing. The method uses SHA-256 for both the mask generation function (MGF) and the message digest.

2.5 Additional Features

Beyond the original implementation, we can include additional features such as dynamic updates. Implementing dynamic updates in a Merkle Tree involves adding the ability to insert and delete data blocks while maintaining the integrity and the correct structure of the tree. This not only changes the tree's structure but also affects proof generation and the verification process. Appendix 7.6 displays the methods added to the implementation to support dynamically adding and removing leaves from the tree.

To add a new leaf to the Merkle Tree, the *insert_data_block* method allows adding new data blocks, updates the tree, and recalculates the root hash. Removing a leaf from the Merkle Tree with the *delete_data_block* method requires the tree to be adjusted and the affected hashes recalculated. Possible improvements on this design are covered in section 5.

3 Performance Evaluation

Evaluating the performance of a Merkle Tree implementation involves analyzing the complexity of three main operations: construction, proof generation, and proof verification. This analysis assumes the tree is balanced, which is typically the case if we're considering perfect binary trees.

3.1 Construction

Building a Merkle Tree involves hashing all the leaves and then iteratively hashing pairs of nodes to build up the tree layers until the root is reached.

If there are n data blocks (leaves), the number of operations in the first level (hashing all leaves) is *O(n)*. Each level reduces the number of items to hash by half. Thus, the number of hashing operations at each level is $n/2, n/4, n/8, \ldots, 1$, whose sum converges to $2n$. Therefore, constructing a Merkle Tree is *O(n)*.

3.2 Proof Generation

Generating a proof for a given leaf involves tracing the path from that leaf to the root and collecting the sibling hash at each level. The height of the tree for n leaves is

$$log_2(n)$$

since each level halves the number of nodes. For each level, one hash pair (the sibling node) is collected, so the number

$$P(r = r') = 2^{-m} + (1-2^{-n})(1-(1-2^{-n})^k)$$

Figure 4: Probability of root collisions given path length and hash size. [2]

of operations is directly proportional to the height of the tree. Therefore, generating a proof of inclusion for any given data block is $O(\log n)$.

3.3 Verification

Verifying a proof involves hashing the data block and then successively hashing it with the sibling hashes provided in the proof until the root hash is reached. Similar to proof generation, this involves hashing at each level from the leaf to the root. As with proof generation, the number of operations is proportional to the tree height,

$$log_2(n)$$

Verifying a proof is $O(\log n)$, as it requires a single hash computation for each level of the tree from the leaf to the root.

4 Security Evaluation of Merkle Trees

4.1 Hash Collisions

The probability of root collisions plays a major role in determining the overall strength of the security of a certain Merkle Tree implementation. This is critical because root collisions in Merkle Trees could potentially undermine the security mechanisms of blockchain networks, leading to vulnerabilities in data integrity and authentication processes.

A study by Kuznetsov et al. [2] on the implications of certain parameters of a Merkle Tree found that the probability of root collisions is not static but depends on the path length k and the hash size m as seen in Formula 4.

This formula demonstrates how the likelihood of collisions increases with the length of the path. This significant finding has substantial implications for the security of blockchain systems utilizing Merkle Trees.

5 Potential Improvements

5.1 Dynamic Updates

Dynamic updates mean the path and the sibling nodes involved in generating proofs for any given node could change. Therefore, after any update, the proofs of remaining nodes must be recalculated if requested. Similarly, for verification, the root hash will change after an update, which means any proofs generated prior to the update will no longer be valid for verification against the new root. It's crucial to re-verify with the new root and updated proofs.

Both insertion and deletion operations require the tree to be rebuilt. Rebuilding the tree every time a node is added or removed might not be efficient for large datasets. This could be optimized by only recalculating the affected branches rather than the entire tree. Efficient re-balancing techniques or incremental updates can be considered.

6 Conclusion

This report began by covering the theoretical foundation of Merkle Trees, including digital signatures, signature amortization, and the structure and operation of Merkle Trees. It then moved on to discuss an implementation of Merkle Trees in Python. After that, it analyzed the implementation in terms of performance and security, along with potential improvements.

References

[1] DIFFIE, W., AND HELLMAN, M. New directions in cryptography. *IEEE Transactions on Information Theory 22* (1976), 644–654.

[2] ET AL., K. Merkle trees in blockchain: A study of collision probability and security implications.

[3] JUNG MIN PARK, E. K. P. C., AND SIEGEL, H. J. Efficient multicast packet authentication using signature amortization. *IEEE* (2002).

[4] MERKLE, R. A digital signature based on a conventional encryption function.

[5] OF STANDARDS, N. I., AND (NIST), T. Proposed digital signature standard (dss). *FIPS Publication* (1993).

[6] WIKIPEDIA. https://commons.wikimedia.org/wiki/file:digital$_{s}$*ignature*$_{d}$*iagram.svg*.

7 Appendix

7.1 Hash Data

```
    def hash_data(data):
        if isinstance(data, str):
            data = data.encode()
        return hashlib.sha256(data).hexdigest()
```

7.2 Tree Construction

```
class MerkleTree:
    def __init__(self):
        self.leaves = []
        self.tree = []
        self.root = None

    def add_data(self, data):
        for block in data:
            self.leaves.append(hash_data(block))

    def build_tree(self):
        current_level = self.leaves[:]

        if len(current_level) % 2 == 1:
            current_level.append(current_level[-1])
        self.tree = [current_level]

        while len(current_level) > 1:
            next_level = []

            for i in range(0, len(current_level), 2):
                hashed_nodes = hash_data(current_level[i] + current_level[i + 1])
                next_level.append(hashed_nodes)

            if len(next_level) % 2 == 1 and len(next_level) > 1:
                next_level.append(next_level[-1])

            self.tree.append(next_level)
            current_level = next_level

        self.root = current_level[0] if current_level else None

                private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048
        )

        public_key = private_key.public_key()

        signature = self.sign_tree(private_key)

        return self.root, signature, public_key
```

7.3 Proof Generation

```
def get_proof(self, data_block):
    block_hash = hash_data(data_block)
    index = self.leaves.index(block_hash)
    proof = []

    for level in self.tree[:-1]:
        other_index = index ^ 1

        if other_index < len(level):
            proof.append((level[other_index], 'L' if other_index < index else 'R'))

        index //= 2
    return proof
```

7.4 Proof Verification

```
def verify_proof(self, data_block, proof, root):
    current_hash = hash_data(data_block)
    for node, position in proof:
        if position == 'L':
            current_hash = hash_data(node + current_hash)
        else:
            current_hash = hash_data(current_hash + node)
    return current_hash == root
```

7.5 Signature Amortization

```
def sign_tree(self, private_key):
    return private_key.sign(
        self.root.encode(),
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

def verify_signature(self, signature, public_key):
    try:
        public_key.verify(
            signature,
            self.root.encode(),
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except Exception:
        return False
```

7.6 Additional Features

```
def insert_data_block(self, data_block):
    """
     Insert a new data block into the tree and rebuild the tree.

     Args:
         data_block (str or bytes): the data block to be added to the tree
     """
    self.leaves.append(hash_data(data_block))
    self.build_tree()

def delete_data_block(self, data_block):
    """
     Delete a data block from the tree and rebuild the tree.

     Args:
         data_block (str or bytes): the data block to be removed from the tree
     """
    try:
        block_hash = hash_data(data_block)
        self.leaves.remove(block_hash)
        self.build_tree()
    except ValueError:
        print("Block not found in the leaves.")
```