Muhammad Ahmad
Bscs22026

Blockchain

Assignment-2

Report

Question no 1:

Implementing Merkle Tree:

```python
def create_merkle_tree(file):
    chunks = []
    tree = []
    hashes = []
    with open(file, 'rb') as f:
        while True:
            chunk = f.read(1024)
            if not chunk:
                break
            chunks.append(chunk)
            hashes.append(calculate_hash(chunk).hexdigest())

    print(f"Total chunks: {len(hashes)}")
    tree.append(hashes)

    while len(hashes) > 1:
        new_hashes = []
        for i in range(0, len(hashes), 2):
            if i + 1 < len(hashes):
                parent = hashes[i] + hashes[i + 1]
                parent_hash = calculate_hash(parent.encode()).hexdigest()
                new_hashes.append(parent_hash)
            else:
                new_hashes.append(hashes[i])

        hashes = new_hashes
        tree.append(hashes)

    print("Merkle Tree built successfully")

    return hashes[0], tree, chunks
```

Its just

taking a file, dividing it in 1KB each and taking it hash using SHA-256 and then storing its hashes. I am also building the tree level by level like in first level there are sole hash and then by making pair of hash after concatinating them I am again taking their hash this will be my second layer If the length was odd I will take the last piece as it is upwords and it will repeat until I have only one hash which is the root of the tree. I am returning the root, tree and chunks.

```
Total chunks: 10250
Merkle Tree built successfully
Merkle Root: dd3e7a9e14bb9a954d116109f39452e7160834bc4736438543351ffb30ab4c15
```

Proving the membership of a specific chunk:

```python
def proof_membership(root, tree, chunk):

    hash_chuck = calculate_hash(chunk).hexdigest()
    proof = []
    proof.append(hash_chuck)
    for i in range(len(tree)):
        index = tree[i].index(hash_chuck)
        if(index % 2 == 0):
            if(index < len(tree[i]) - 1):
                hash_chuck = hash_chuck + tree[i][index + 1]
                hash_chuck = calculate_hash(hash_chuck.encode()).hexdigest()
        else:
            hash_chuck = tree[i][index - 1] + hash_chuck
            hash_chuck = calculate_hash(hash_chuck.encode()).hexdigest()

        proof.append(hash_chuck)


    return (proof[-1] == root)
```

Now I will calculate the hash of the specific chunk and will find in my last level of tree then I will check its index if its even then I will have to combine it with next index if its last then it will be on next level, if its odd I will just combine it with previous chunk. I will

repeat these steps until I reach the root of the tree. Then I will just compare both root If they are equal chunk is present in tree and vive versa.

Proving non-Membership of a chunk:

```python
def proof_non_membership(root, tree, chunk, chunks):

    chunks.sort();
    for i in range(len(chunks)):
        if chunks[i] < chunk < chunks[i+1]:
            if(proof_membership(root, tree, chunks[i])):
                if(proof_membership(root, tree, chunks[i+1])):
                    print('The chunk is not in the tree')
                    return True

    print("The chunk is outside of the known range.")
    return True
```

Now to check if a chunk is not present in tree we will sort the chunks and will prove a chunk is present that should come before it and will prove a chunk that should come after it. This is how we can prove non-Membership of a chunk.

```python
def main():

    file = 'text.txt'

    root, tree, chunks = create_merkle_tree(file)

    print("Merkle Root:", root)

    print(f"Proving membership of chunk: {chunks[2]}")

    if(proof_membership(root, tree, chunks[2])):
        print('The chunk is in the tree')
    else:
        print('The chunk is not in the tree')


    ch = b'Just a chunk to check if chunk is not in tree'
    print(f"Proving non-membership of chunk: {ch}")
    proof_non_membership(root, tree, ch, chunks)
```

File Generator:

```python
def generate_textfile(path, size):
    bytes = size * 1024 * 1024
    sz = 0
    with open(path, 'w') as file:
        while sz < bytes:
            text =''.join(random.choices(string.ascii_letters + string.digits, k=1024))
            file.write(text + '\n')
            sz += len(text)

    print(f"Text file '{path}' created with {size} MB of data.")
```

Question no 2:

Merkle Tree:

A **Merkle tree** is a binary tree used to verify large datasets' integrity by breaking them into smaller "chunks." Each leaf node in the tree contains the hash of a chunk of data, and parent nodes contain the hash of their two child nodes, eventually leading to a single **Merkle root** at the top.

Key Features:

- Efficient for verifying the inclusion of data blocks (Proof of Membership).
- Works best when the number of leaf nodes is a power of two.
- Requires multiple layers of hashing, which may increase storage and computation overhead for large datasets.

Verkle Tree:

A **Verkle tree** is an advanced data structure that combines a **vector commitment** with a tree structure to optimize the storage of blockchain data. It's a more memory-efficient alternative to the Merkle tree, designed to handle larger datasets with lower storage and computational costs.

Key Features:

- Uses polynomial commitments instead of binary hashes, allowing for much smaller proof sizes so it is memory efficient.
- Verkle trees significantly reduce the size of proofs required to verify membership, leading to faster blockchain verification.
- Nodes have more branches compared to Merkle trees.

Question no 3:

1. Proof of Work (PoW):

Miners compete to solve complex cryptographic puzzles, and the first to solve it gets to add a new block to the blockchain and is rewarded with cryptocurrency.

Key Characteristics:

- Requires significant computational power and energy.
- Resistant to Sybil attacks but vulnerable to 51% attacks.
- Used in cryptocurrencies like Bitcoin and Ethereum (pre-Ethereum 2.0).

2. Proof of Stake (PoS):

Validators (instead of miners) are chosen to create new blocks based on the number of coins they "stake" (lock up) as collateral. The more tokens you hold, the higher the chance you are chosen to validate a block.

Key Characteristics:

- Requires less computational power compared to PoW.
- The risk is that a participant who tries to attack the system loses their staked coins.
- Used in Ethereum 2.0, Cardano, and Tezos.

3. Proof of Ownership (PoO):

Verifies the ownership of assets by requiring participants to prove they control specific assets or resources, either digital or physical.

Key Characteristics:

- Focuses on establishing control over assets rather than solving puzzles or staking tokens.

- Used in some blockchain applications for digital ownership verification.

4. Proof of Authority (PoA):

A limited number of pre-approved validators (authorities) are responsible for validating transactions. These validators are selected based on their reputation and are typically known entities.

Key Characteristics:

- More centralized compared to PoW and PoS, as it relies on a few trusted validators.
- Faster transaction speeds due to fewer validators.
- Used in private or consortium blockchains like VeChain and some Ethereum test networks (e.g., Ropsten).

5. Delegated Proof of Stake (DPoS):

Token holders vote for a small group of delegates who are responsible for validating transactions and securing the network. Delegates are rotated regularly based on voting.

Key Characteristics:

- Stakeholders vote on who gets to validate transactions.
- Fewer validators means faster consensus and lower resource requirements.
- Used in blockchains like EOS, TRON, and Lisk.

6. Proof of Burn (PoB):

Participants "burn" (destroy) their tokens by sending them to an irrecoverable address. In return, they are granted the right to mine or validate transactions.

Key Characteristics:

- Burning tokens reduces the total supply, potentially increasing the value of remaining tokens.
- Participants have an economic incentive to act honestly because they’ve lost tokens permanently.
- Used in some experimental blockchains but not widely adopted.