EXPERIMENT 1

Aim

Write a Python program to understand SHA and Cryptography in Blockchain, Merkle root tree hash.

Tasks Performed:

1. Hash Generation using SHA-256: Developed a Python program to compute a SHA-256 hash for any given input string using the hashlib library.

2. Target Hash Generation with Nonce: Created a program to generate a hash code by concatenating a user input string and a nonce value to simulate the mining process.

3. Proof-of-Work Puzzle Solving: Implemented a program to find the nonce that, when combined with a given input string, produces a hash starting with a specified number of leading zeros.

4. Merkle Tree Construction: Built a Merkle Tree from a list of transactions by recursively hashing pairs of transaction hashes, doubling up last nodes if needed, and generated the Merkle Root hash for blockchain transaction integrity.

Theory

Cryptographic Hash Functions in Blockchain

Cryptographic hash functions are the foundation of blockchain security. A hash function takes an input of any size and produces a fixed-length output known as a hash value. In blockchain systems, SHA-256 (Secure Hash Algorithm) is commonly used due to its strong security properties such as determinism, collision resistance, and irreversibility. Even a small change in input data results in a completely different hash, ensuring data integrity and tamper detection. Hash functions link blocks together and protect transaction data from unauthorized modification.

Merkle Tree

A Merkle Tree (also known as a hash tree) is a data structure used in blockchain to efficiently organize and verify large sets of transactions. It allows multiple transaction hashes to be combined into a single hash called the Merkle Root. Merkle Trees play a crucial role in ensuring data integrity and efficient verification of transactions within a block.

Structure of a Merkle Tree

A Merkle Tree is a **binary tree structure** used to organize and verify data efficiently. Its structure consists of multiple levels:

1. **Leaf Nodes**
    - The bottom level of the tree
    - Each leaf node contains the **hash of a transaction or data block**

2. **Intermediate (Parent) Nodes**
    - Formed by **combining and hashing two child node hashes**
    - Each parent node stores the hash of its children

3. **Handling Odd Number of Nodes**
    - If there is an odd number of hashes at any level, the **last hash is duplicated**
    - This ensures every node has a pair for hashing

4. **Root Node (Merkle Root)**
    - The topmost node of the tree
    - Represents the **single hash value summarizing all transactions**

Merkle Root

The Merkle Root is the final hash obtained at the top of the Merkle Tree. It represents a single cryptographic summary of all transactions in a block. Any modification to a transaction changes its hash, which in turn changes all parent hashes up to the Merkle Root. Therefore, the Merkle Root guarantees the integrity and authenticity of all transactions in a block.

Working of Merkle Tree

The working of a Merkle Tree involves the following steps:

1. Each transaction is hashed using a cryptographic hash function.
2. Hashes are paired and concatenated.
3. The concatenated values are hashed again to form parent nodes.
4. This process repeats recursively until one hash remains.
5. The final hash obtained is the Merkle Root.

This hierarchical hashing enables efficient verification of transactions without needing to process all data.

Benefits of Merkle Tree

Merkle Trees offer several advantages:

- Efficient verification of large data sets
- Reduced storage requirements
- Fast transaction validation
- Improved data integrity and security
- Enables lightweight clients to verify transactions without downloading the entire blockchain

Uses of Merkle Tree in Blockchain

In blockchain systems, Merkle Trees are used to:

- Organize and validate transactions within a block
- Detect data tampering quickly
- Support Simplified Payment Verification (SPV)
- Reduce bandwidth and storage usage
- Maintain trustless verification in decentralized networks

Use Cases of Merkle Tree

Merkle Trees are widely used in:

- Bitcoin and other cryptocurrency blockchains
- Distributed systems for data verification
- Secure file systems
- Peer-to-peer networks
- Version control systems (e.g., Git)
- Data synchronization and integrity checking

Code

Part 1: SHA-256 Hash Generation

```
import hashlib
data = input("Enter the input string: ")
hash_value = hashlib.sha256(data.encode()).hexdigest()
print("SHA-256 Hash:", hash_value)
```

Part 2: Hash Generation with Nonce

```
import hashlib
data = input("Enter the input string: ")
nonce = int(input("Enter nonce value: "))
combined_data = data + str(nonce)
hash_with_nonce = hashlib.sha256(combined_data.encode()).hexdigest()
print("Hash with Nonce:", hash_with_nonce)
```

Part 3: Proof-of-Work Puzzle Solving

```
import hashlib
data = input("Enter the input string: ")
difficulty = int(input("Enter number of leading zeros required: "))
prefix = '0' * difficulty
nonce = 0
while True:
    text = data + str(nonce)
    hash_result = hashlib.sha256(text.encode()).hexdigest()
    if hash_result.startswith(prefix):
        break
    nonce += 1
print("Nonce found:", nonce)
print("Valid Proof-of-Work Hash:", hash_result)
```

Part 4: Merkle Tree Construction

```
import hashlib
def sha256(data):
    return hashlib.sha256(data.encode()).hexdigest()
def merkle_root(transactions):
    hashes = [sha256(tx) for tx in transactions]
    while len(hashes) > 1:
        if len(hashes) % 2 != 0:
            hashes.append(hashes[-1])
        new_level = []
```

```
        for i in range(0, len(hashes), 2):
            combined_hash = hashes[i] + hashes[i + 1]
            new_level.append(sha256(combined_hash))
        hashes = new_level
    return hashes[0]
transactions = [
    "Alice pays Bob 10 BTC",
    "Bob pays Charlie 5 BTC",
    "Charlie pays Dave 2 BTC",
    "Dave pays Eve 1 BTC"
]
print("Merkle Root Hash:", merkle_root(transactions))
```

Output

Part 1: SHA-256 Hash Generation

```
Enter the input string: a
SHA-256 Hash: ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
```

Part 2: Hash Generation with Nonce

```
Enter the input string: ab
Enter nonce value: 70158
Hash with Nonce: 09e3960765a97b3443fb54662d8e4511b0e6fb194780214e89dcedff0b1aa84e
```

Part 3: Proof-of-Work Puzzle Solving

```
Enter the input string: a
Enter number of leading zeros required: 5
Nonce found: 862183
Valid Proof-of-Work Hash: 000008a7d97881343ade2a4fa70a5b71af75eacf90c87f173bc96847ba42c38d
```

Part 4: Merkle Tree Construction

```
Merkle Root Hash: 971669fd8fe0f5915600b460f2751cd03a4f82e2a16b7f8b154f2b3ca31500b8
```

Conclusion

SHA-256 hash generation, nonce-based hash computation, Proof-of-Work puzzle solving, and Merkle Tree construction were successfully implemented using Python, demonstrating fundamental blockchain mechanisms.