

Exp 1

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

Theory:

1. Cryptographic Hashing

Cryptographic hashing is a mathematical transformation that converts an input of any size into a unique, fixed-size string of bytes. Unlike standard encryption, hashing is a trapdoor function, meaning it is designed to be a one-way street: easy to calculate in one direction but computationally impossible to reverse.

In the context of data structures, hashing acts as a "digital lock." If a single bit of the input data is altered, the resulting hash changes so drastically that the two outputs appear entirely unrelated. This property is used to verify the integrity of everything from passwords and software downloads to multi-billion dollar financial transactions on a blockchain.

Characteristics of Cryptographic Hash Functions

1. Deterministic – Same input always produces the same hash.
2. Fixed output length – Regardless of input size.
3. Fast computation – Efficient to compute hash.
4. Pre-image resistance – Hard to find original input from hash.
5. Collision resistance – Hard to find two different inputs with the same hash.
6. Avalanche effect – Small change in input results in a large change in hash.

Examples of Hash Functions

- SHA-256 (used in Bitcoin)
- SHA-1
- MD5 (not secure now)

Purpose of Cryptographic Hashing

- Data integrity verification
- Digital signatures
- Password storage
- Blockchain security

2. Properties of SHA-256

The Secure Hash Algorithm 256-bit (SHA-256) is a member of the SHA-2 family designed by the NSA. It produces a 256-bit (32-byte) signature. Its specific properties include:

- Deterministic: Input X always results in Hash $H(X)$. This is critical for distributed systems to reach consensus.
- Pre-image Resistance: It is mathematically infeasible to find the input X from its hash $H(X)$. This prevents hackers from "reverse-engineering" sensitive data.
- Second Pre-image Resistance: Given an input X , it is impossible to find another input Y such that $H(X) = H(Y)$.
- Collision Resistance: While theoretically possible (since there are infinite inputs and finite outputs), the odds of finding two different inputs that produce the same SHA-256 hash are 1 in 2^{256} , a number larger than the atoms in the observable universe.
- Avalanche Effect: A minor change (like adding a period at the end of a sentence) flips roughly 50% of the bits in the resulting hash, ensuring no leakage of information about the original data.

3. Merkle Tree (Hash Tree)

A Merkle Tree is a specialized data structure, typically a binary tree, where every leaf node contains the cryptographic hash of a data block, and every non-leaf node contains the cryptographic hash of its child nodes' combined labels.

Invented by Ralph Merkle in 1979, the tree allows for the summarization of large datasets. Instead of requiring a system to hold an entire database to verify a single record, the system only needs to store the "path" leading up to the root. This transforms a linear search process into a logarithmic one, drastically reducing the computational overhead for data verification.

Components of a Merkle Tree

1. Leaf Nodes

- Contain a hash of individual data blocks.
- Example: $H1 = \text{Hash}(\text{Data1})$

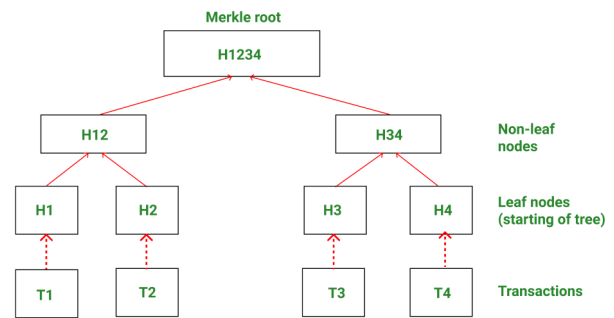
2. Intermediate (Internal) Nodes

- Store hash of concatenation of child hashes.
- Example: $H12 = \text{Hash}(H1 + H2)$

3. Merkle Root

- The final top hash representing all data.

Example Structure



4. Merkle Root

The Merkle Root is the single hash at the very top of the tree (the "apex"). It serves as the ultimate summary of every piece of data contained within the tree.

In blockchain technology, the Merkle Root is stored in the Block Header. This is vital for security: because the root is derived from the hashes of all transactions in the block, the root itself becomes a "commitment" to those transactions. If a single transaction is modified, the change ripples up the tree, resulting in a different Merkle Root, which would then invalidate the block header and be rejected by the network.

5. Working of Merkle Tree

The construction of a Merkle Tree is a recursive process of "Hash, Pair, and Re-hash":

1. Hashing Leaf Nodes: Data blocks (e.g., Transactions \$A, B, C, D\$) are hashed individually: $\$H_A, H_B, H_C, H_D\$$.
2. Pairing: Hashes are paired together. $\$H_A\$$ and $\$H_B\$$ become a pair; $\$H_C\$$ and $\$H_D\$$ become a pair.
3. Concatenation: The strings of the two hashes are concatenated (joined).
4. Hashing the Parent: The concatenated string is hashed again to create the parent node: $\$H_AB\} = \text{Hash}(H_A + H_B)\$$.
5. Reaching the Root: This continues until only one hash remains.
 - Note: If the number of transactions is odd, the last transaction hash is duplicated to form a pair, ensuring the tree remains balanced.

6. Benefits of Merkle Tree: Efficiency and Verification

The primary benefit is the ability to provide a Merkle Proof.

- Logarithmic Scaling: In a block with n transactions, you only need $\log_2(n)$ hashes to prove a transaction's existence. For 1 million transactions, you only need to check 20 hashes.
- Minimal Data Transfer: Servers can prove to a client that a file is uncorrupted without sending the entire file. They only send the small "audit path" of hashes.
- Separation of Concerns: It allows for the existence of "Light Nodes" (like mobile crypto wallets) that don't have the storage capacity for the full blockchain but can still verify transactions with 100% certainty.

7. Use of Merkle Tree in Blockchain

Blockchain utilizes Merkle Trees to solve the problem of scalability vs. security.

- Block Headers: By placing the Merkle Root in the block header, the header stays a constant size (roughly 80 bytes in Bitcoin), regardless of whether the block contains 5 or 5,000 transactions.
- Tamper-Proofing: It links the transaction data to the "Chain of Proof." Since each block header contains the hash of the previous block header, the Merkle Root binds the transactions into the immutable history of the entire chain.

- Simplified Payment Verification (SPV): This is the mechanism that allows a user to verify their payment went through by asking a full node for just the Merkle Path, rather than downloading the entire ledger.

8. Use Cases of Merkle Tree

Application	How it uses Merkle Trees
Bitcoin/Ethereum	To summarize transactions within a block and enable SPV clients.
Git (Version Control)	To identify file changes. Each commit hash is essentially a Merkle Root of the file directory.
ZFS File System	To detect "Bit Rot" (data corruption) on physical disks and automatically repair it.
NoSQL Databases	Systems like Apache Cassandra use them to compare data between different nodes in a cluster to find where they are out of sync.
IPFS	The Interplanetary File System uses Merkle DAGs to create a content-addressed web where the URL is the hash of the content.

Colab Notebook: <https://colab.research.google.com/drive/1dV79RjwqnHb9vEjT3KESUi1DzJujf-ci?usp=sharing>

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.

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

Enter the input string: Shreyash

SHA-256 Hash: f4553f8d6c9129555faba73f4d8ac1434115bd9b32620a271f32298e5939e660

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.

```
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)
```

Enter the input string: Shreyash Dhekane

Enter nonce value: 3

Hash with Nonce: 4b586830ea80c2f9e23b1fc80d5d17933f2b2e68b972a3e5bf827899ac6a0f30

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.

```
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)
Enter the input string: Shreyash Dhekane
Enter number of leading zeros required: 3
Nonce found: 1289
Valid Proof-of-Work Hash: 0004bfb9b629566bbd8d499afa7e26ccacdfc4216e068705db2c24f7b1351815
```

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.

```
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]) # duplicate last hash if odd
        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]

# ♦ Unique non-BTC transactions
transactions = [
    "Order #1023: Laptop purchased by Rahul",
    "Invoice #559: Payment received from Ananya",
    "Library system: Book issued to Student ID 231",
    "Online exam portal: Answer sheet submitted",
    "Electricity board: Bill paid for Meter 77891"
]

print("Merkle Root Hash:", merkle_root(transactions))

Merkle Root Hash: f8f698c1e2c7cfc6f49408581f38a8f186c22bd034176e1bba6cf1594fc93386
```

Conclusion:-

Cryptographic hash functions like SHA-256 help keep blockchain data safe, secure, and unchangeable. Merkle Trees organize transactions in a way that makes it easy to verify them quickly and detect any tampering. Together, they make blockchain trustworthy, capable of handling large amounts of data without compromising security. These concepts help us understand how blockchain maintains security, and efficiency even with large amounts of data.