1. **MERKLE TREE**

import hashlib

def construct\_merkle\_tree(transactions):

    if len(transactions) == 0:

        return None

    if len(transactions) == 1:

        return transactions[0]

    # Recursively construct Merkle tree

    new\_transactions = []

    for i in range(0, len(transactions)-1, 2):

        combined\_hash = hashlib.sha256((transactions[i] + transactions[i+1]).encode()).hexdigest()

        new\_transactions.append(combined\_hash)

    # If the number of transactions is odd, hash the last transaction with itself

    if len(transactions) % 2 == 1:

        combined\_hash = hashlib.sha256((transactions[-1] + transactions[1]).encode()).hexdigest()

        new\_transactions.append(combined\_hash)

    return construct\_merkle\_tree(new\_transactions)

# Take transactions as input from the user

def take\_transactions\_input():

    transactions = []

    num\_transactions = int(input("Enter the number of transactions: "))

    for i in range(num\_transactions):

        transaction = input("Enter transaction " + str(i+1) + ": ")

        transactions.append(transaction)

    return transactions

# Test Merkle tree construction with user input

transactions = take\_transactions\_input()

merkle\_root = construct\_merkle\_tree(transactions)

print("Merkle Root:", merkle\_root)

1. **PROOF OF WORK / MINING**

import hashlib

def proof\_of\_work(difficulty):

    block\_header = input("Enter the block header: ")

    target = '0' \* difficulty

    nonce = 0

    while True:

        block\_hash = hashlib.sha256((block\_header + str(nonce)).encode()).hexdigest()

        if block\_hash[:difficulty] == target:

            return nonce, block\_hash

        nonce += 1

# Implement proof-of-work with user input

difficulty = int(input("Enter the mining difficulty: "))

nonce, block\_hash = proof\_of\_work(difficulty)

print("Nonce:", nonce)

print("Block Hash:", block\_hash)

1. **GENESIS BLOCK**

import hashlib

def create\_genesis\_block(transactions):

    merkle\_root = construct\_merkle\_tree(transactions)

    genesis\_block = {

        'previous\_block\_hash': '0',  # Previous block hash is usually set to '0' for the genesis block

        'merkle\_root': merkle\_root,

        'nonce': 0,  # Nonce for the genesis block can be set to any value

        'transactions': transactions

    }

    return genesis\_block

def construct\_merkle\_tree(transactions):

    if len(transactions) == 0:

        return None

    if len(transactions) == 1:

        return transactions[0]

    # Recursively construct Merkle tree

    new\_transactions = []

    for i in range(0, len(transactions)-1, 2):

        combined\_hash = hashlib.sha256((transactions[i] + transactions[i+1]).encode()).hexdigest()

        new\_transactions.append(combined\_hash)

    # If the number of transactions is odd, hash the last transaction with itself

    if len(transactions) % 2 == 1:

        combined\_hash = hashlib.sha256((transactions[-1] + transactions[-1]).encode()).hexdigest()

        new\_transactions.append(combined\_hash)

    return construct\_merkle\_tree(new\_transactions)

# Take transactions as input from the user

def take\_transactions\_input():

    transactions = []

    num\_transactions = int(input("Enter the number of transactions: "))

    for i in range(num\_transactions):

        transaction = input("Enter transaction " + str(i+1) + ": ")

        transactions.append(transaction)

    return transactions

# Create genesis block with user input

genesis\_transactions = take\_transactions\_input()

genesis\_block = create\_genesis\_block(genesis\_transactions)

print("Genesis Block:", genesis\_block)

**HASHING AND DIGITAL SIGNATURE:**

package javaapplication2;

import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.MessageDigest;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.security.Signature;

import java.util.Scanner;

public class JavaApplication2 {

public static void main(String[] args) throws Exception {

Scanner scanner = new Scanner(System.in);

// Take input from the user

System.out.print("Enter message to be hashed and signed: ");

String message = scanner.nextLine();

// Test Hashing

String hash = HashingLab.calculateHash(message);

System.out.println("Hash of input data: " + hash);

// Test Digital Signature

KeyPair keyPair = SignatureLab.generateKeyPair();

byte[] signature = SignatureLab.signMessage(hash, keyPair.getPrivate());

// Simulate tampering with the message

// message = "This is a tampered message";

// Hash the tampered message

String tamperedHash = HashingLab.calculateHash(message);

// Verify Signature

boolean isVerified = SignatureLab.verifySignature(tamperedHash, signature, keyPair.getPublic());

System.out.println("Is signature verified? " + isVerified);

scanner.close();

}

}

class HashingLab {

public static String calculateHash(String input) throws Exception {

MessageDigest digest = MessageDigest.getInstance("SHA-256");

byte[] hashBytes = digest.digest(input.getBytes());

// Convert byte array to hexadecimal string

StringBuilder hexString = new StringBuilder();

for (byte b : hashBytes) {

String hex = Integer.toHexString(0xff & b);

if (hex.length() == 1)

hexString.append('0');

hexString.append(hex);

}

return hexString.toString();

}

}

class SignatureLab {

public static KeyPair generateKeyPair() throws Exception {

KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");

keyGen.initialize(2048);

return keyGen.generateKeyPair();

}

public static byte[] signMessage(String message, PrivateKey privateKey) throws Exception {

Signature signature = Signature.getInstance("SHA256withRSA");

signature.initSign(privateKey);

signature.update(message.getBytes());

return signature.sign();

}

public static boolean verifySignature(String message, byte[] signature, PublicKey publicKey) throws Exception {

Signature verifier = Signature.getInstance("SHA256withRSA");

verifier.initVerify(publicKey);

verifier.update(message.getBytes());

return verifier.verify(signature);

}

}

**BITCOIN MINING SIMULATION:**

class Block:

    def \_\_init\_\_(self, version, previous\_block\_hash, timestamp, merkle\_root, nonce):

        self.version = version

        self.previous\_block\_hash = previous\_block\_hash

        self.timestamp = timestamp

        self.merkle\_root = merkle\_root

        self.nonce = nonce

def create\_block():

    version = input("Enter block version: ")

    previous\_block\_hash = input("Enter previous block hash: ")

    timestamp = input("Enter timestamp: ")

    merkle\_root = input("Enter Merkle root: ")

    nonce = input("Enter nonce: ")

    return Block(version, previous\_block\_hash, timestamp, merkle\_root, nonce)

# Create a simulated blockchain

def create\_blockchain():

    blockchain = []

    num\_blocks = int(input("Enter the number of blocks in the blockchain: "))

    for i in range(num\_blocks):

        print("Enter details for Block", i+1)

        block = create\_block()

        blockchain.append(block)

    return blockchain

blockchain = create\_blockchain()

for i, block in enumerate(blockchain):

    print("\nBlock", i+1, "Details:")

    print("Version:", block.version)

    print("Previous Block Hash:", block.previous\_block\_hash)

    print("Timestamp:", block.timestamp)

    print("Merkle Root:", block.merkle\_root)

    print("Nonce:", block.nonce)

**MINING OPTIMIZATION:**

import hashlib

import multiprocessing

import time

def proof\_of\_work(difficulty, block\_header, start\_nonce, end\_nonce, result\_queue):

    target = '0' \* difficulty

    nonce = start\_nonce

    while nonce < end\_nonce:

        block\_hash = hashlib.sha256((block\_header + str(nonce)).encode()).hexdigest()

        if block\_hash[:difficulty] == target:

            result\_queue.put((nonce, block\_hash))

            return

        nonce += 1

    result\_queue.put(None)

def mine\_with\_parallel\_processing(difficulty, block\_header, num\_processes):

    start\_time = time.time()

    result\_queue = multiprocessing.Queue()

    processes = []

    for i in range(num\_processes):

        start\_nonce = i \* (2\*\*32 // num\_processes)

        end\_nonce = (i + 1) \* (2\*\*32 // num\_processes)

        process = multiprocessing.Process(target=proof\_of\_work, args=(difficulty, block\_header, start\_nonce, end\_nonce, result\_queue))

        processes.append(process)

        process.start()

    for process in processes:

        process.join()

    while not result\_queue.empty():

        result = result\_queue.get()

        if result:

            nonce, block\_hash = result

            end\_time = time.time()

            print("Nonce:", nonce)

            print("Block Hash:", block\_hash)

            print("Time taken:", end\_time - start\_time, "seconds")

            return

def mine\_with\_nonce\_selection(difficulty, block\_header, start\_nonce, end\_nonce):

    start\_time = time.time()

    nonce = start\_nonce

    while nonce < end\_nonce:

        block\_hash = hashlib.sha256((block\_header + str(nonce)).encode()).hexdigest()

        if block\_hash[:difficulty] == '0' \* difficulty:

            end\_time = time.time()

            print("Nonce:", nonce)

            print("Block Hash:", block\_hash)

            print("Time taken:", end\_time - start\_time, "seconds")

            return

        nonce += 1

# Example of mining with parallel processing

difficulty = 4  # Adjust difficulty as needed

block\_header = "PreviousBlockHash:Transaction1:Transaction2:Transaction3:Nonce:"

num\_processes = 4  # Adjust number of processes as needed

mine\_with\_parallel\_processing(difficulty, block\_header, num\_processes)

# Example of mining with nonce selection strategy

start\_nonce = 0

end\_nonce = 2\*\*32

mine\_with\_nonce\_selection(difficulty, block\_header, start\_nonce, end\_nonce)

**CUSTOM SCRIPTS:**

import binascii

import hashlib

# Define the Bitcoin message prefix

BITCOIN\_MESSAGE\_PREFIX = '\x18Bitcoin Signed Message:\n'

# Define the Bitcoin curve parameters

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F

n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

G = (0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798, 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8)

# Define the Bitcoin public key hash

public\_key\_hash = binascii.unhexlify('4b3518229b0d3554fe7cd3796ade632aff3069d8')

# Define the Bitcoin script

script = Script()

script.push\_data(binascii.unhexlify('76a914'))

script.push\_data(public\_key\_hash)

script.push\_data(binascii.unhexlify('88ac'))

# Define the Bitcoin message

message = BITCOIN\_MESSAGE\_PREFIX + binascii.hexlify(script.encode()).decode()

# Define the Bitcoin private key

private\_key = 0x18E14A7B6A307F426A94F8114701E7C8E774E7F9A47E2C2035DB29A206321725

# Define the Bitcoin signature

R = 0x304502210086C8E8E203D1B1CC2B2E8F8A5B4E5E34D8D3B482E8F510C8E1E4A994F1E6DF02274C3D3B0B42F67C129E29A2881139B7E29E0A291119F1E112E89C0B00000000

signature = binascii.unhexlify(R)

# Verify the Bitcoin signature

digest = hashlib.sha256(hashlib.sha256(message.encode()).digest()).digest()

r = int.from\_bytes(signature[:32], byteorder='big')

s = int.from\_bytes(signature[32:], byteorder='big')

if (r > (n + 1) // 2) or (r < 1) or (s > (n + 1) // 2) or (s < 1):

    print('Invalid signature: r or s is out of range.')

elif (pow(r, 2, p) - pow(s, 2, p)) % n != int.from\_bytes(digest,

byteorder='big'):

    print('Invalid signature: signature verification failed.')

else:

    print('Valid signature.')