SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 3rd semester

Blockchain Technology (PGI20G01J)- Lab Manual

Lab 1: Demonstrating Secret Key Cryptography Techniques

Title: Lab 1: Demonstrating Secret Key Cryptography Techniques

Aim: To understand the fundamental concepts of secret key (symmetric) cryptography and implement a basic symmetric encryption/decryption algorithm.

Procedure:

1. **Understand Symmetric Cryptography:** Learn about the principles where the same key is used for both encryption and decryption.
2. **Choose a Simple Cipher:** Select a basic symmetric cipher for implementation, such as the Caesar Cipher.
3. **Implement Encryption:** Write code to encrypt a plaintext message using the chosen cipher and a secret key.
4. **Implement Decryption:** Write code to decrypt the ciphertext back to plaintext using the same secret key.
5. **Test and Verify:** Test the implementation with various inputs and ensure correct encryption and decryption.

Source Code (Python - Caesar Cipher Example):

```
def encrypt_caesar(text, key):
    result = ""
    for char in text:
        if char.isalpha():
            start = ord('a') if char.islower() else ord('A')
            shifted_char = chr((ord(char) - start + key) % 26 + start)
            result += shifted_char
        else:
            result += char
    return result

def decrypt_caesar(text, key):
    return encrypt_caesar(text, -key) # Decrypt by encrypting with negative
key

# Example Usage
if __name__ == "__main__":
    message = "Hello, World!"
    secret_key = 3

    encrypted_message = encrypt_caesar(message, secret_key)
    print(f"Original Message: {message}")
    print(f"Secret Key: {secret_key}")
```

```
    print(f"Encrypted Message: {encrypted_message}")

    decrypted_message = decrypt_caesar(encrypted_message, secret_key)
    print(f"Decrypted Message: {decrypted_message}")
```

Input:

```
Original Message: "Hello, World!"
Secret Key: 3
```

Expected Output:

```
Original Message: Hello, World!
Secret Key: 3
Encrypted Message: Kello, Zruog!
Decrypted Message: Hello, World!
```

Lab 2: Demonstrating Public Key Cryptography Techniques

Title: Lab 2: Demonstrating Public Key Cryptography Techniques

Aim: To understand the principles of public key (asymmetric) cryptography, including key pair generation, and the use of public and private keys for encryption and decryption.

Procedure:

1. **Understand Asymmetric Cryptography:** Learn about the concept of separate public and private keys for encryption and decryption.
2. **Conceptual Key Generation:** Understand how a pair of mathematically linked keys (public and private) are generated.
3. **Conceptual Encryption:** Grasp how a message is encrypted using the recipient's public key.
4. **Conceptual Decryption:** Understand how the encrypted message can only be decrypted by the recipient's corresponding private key.
5. **Discuss Algorithms:** Briefly research and discuss common public key algorithms like RSA or ECC.

Source Code (Python - Conceptual Example using a simplified "key" idea):

```
# This is a highly simplified conceptual example to illustrate the idea of
public/private keys.
# It does NOT implement actual cryptographic security.
class AsymmetricCryptoSimulator:
    def __init__(self):
        # In a real system, these would be complex mathematical values
        self.private_key = "my_secret_private_key_123"
        self.public_key = "my_public_key_abc"

    def encrypt(self, message, recipient_public_key):
        if recipient_public_key == self.public_key:
            print(f"Encrypting '{message}' with recipient's public key:
{recipient_public_key}")
            # Simulate encryption (e.g., reverse string for demonstration)
            return message[::-1] + "_encrypted"
        else:
            return "Error: Invalid public key for encryption."

    def decrypt(self, encrypted_message, private_key):
        if private_key == self.private_key:
            print(f"Decrypting '{encrypted_message}' with private key:
{private_key}")
            # Simulate decryption (e.g., reverse string back)
            if encrypted_message.endswith("_encrypted"):
                return encrypted_message[:-len("_encrypted")][::-1]
            else:
                return "Decryption failed: Not a valid encrypted message."
        else:
            return "Error: Invalid private key for decryption."

# Example Usage
if __name__ == "__main__":
    alice = AsymmetricCryptoSimulator()
    bob = AsymmetricCryptoSimulator() # Bob would have his own keys

    message_to_bob = "Secret message for Bob"

    # Alice encrypts message using Bob's public key (conceptually)
    # For this simulation, we'll use Alice's own public key for simplicity
```

```
    # In a real scenario, Alice would get Bob's public key.
    encrypted_by_alice = alice.encrypt(message_to_bob, alice.public_key)
    print(f"Encrypted by Alice: {encrypted_by_alice}")

    # Bob decrypts message using his private key (conceptually)
    decrypted_by_bob = alice.decrypt(encrypted_by_alice, alice.private_key)
    print(f"Decrypted by Bob: {decrypted_by_bob}")

    # Attempt decryption with wrong key
    print("\nAttempting decryption with wrong key:")
    wrong_key_decryption = alice.decrypt(encrypted_by_alice, "wrong_key")
    print(f"Decryption with wrong key: {wrong_key_decryption}")
```

Input:

```
Message to be encrypted: "Secret message for Bob"
Alice's Public Key: "my_public_key_abc" (used for encryption)
Alice's Private Key: "my_secret_private_key_123" (used for decryption)
```

Expected Output:

```
Encrypting 'Secret message for Bob' with recipient's public key:
my_public_key_abc
Encrypted by Alice: boB rof egassem terceS_encrypted
Decrypting 'boB rof egassem terceS_encrypted' with private key:
my_secret_private_key_123
Decrypted by Bob: Secret message for Bob

Attempting decryption with wrong key:
Decryption with wrong key: Error: Invalid private key for decryption.
```

Lab 3: Demonstrating Hashing Techniques (SHA and MD5)

Title: Lab 3: Demonstrating Hashing Techniques (SHA and MD5)

Aim: To understand the concept of cryptographic hashing and implement the use of common hashing algorithms like SHA-256 and MD5 in Python.

Procedure:

1. **Understand Hashing:** Learn about hash functions, their properties (one-way, collision resistance, fixed-size output), and their use in data integrity.
2. **Import `hashlib`:** Familiarize yourself with Python's built-in `hashlib` module.
3. **Implement MD5 Hashing:** Write code to compute the MD5 hash of a given input string.
4. **Implement SHA-256 Hashing:** Write code to compute the SHA-256 hash of the same input string.
5. **Observe Outputs:** Compare the outputs for different algorithms and understand why SHA-256 is generally preferred over MD5 for security-critical applications.

Source Code (Python):

```python
import hashlib

def calculate_md5_hash(data):
    """Calculates the MD5 hash of the given data."""
    md5_hash = hashlib.md5()
    md5_hash.update(data.encode('utf-8')) # Encode data to bytes
    return md5_hash.hexdigest()

def calculate_sha256_hash(data):
    """Calculates the SHA-256 hash of the given data."""
    sha256_hash = hashlib.sha256()
    sha256_hash.update(data.encode('utf-8')) # Encode data to bytes
    return sha256_hash.hexdigest()

# Example Usage
if __name__ == "__main__":
    input_string = "Hello Blockchain World!"

    md5_result = calculate_md5_hash(input_string)
    sha256_result = calculate_sha256_hash(input_string)

    print(f"Original String: '{input_string}'")
    print(f"MD5 Hash:        {md5_result}")
    print(f"SHA-256 Hash:    {sha256_result}")

    # Demonstrate sensitivity to small changes
    input_string_modified = "Hello Blockchain World." # Changed '!' to '.'
    sha256_modified_result = calculate_sha256_hash(input_string_modified)
    print(f"\nOriginal String (modified): '{input_string_modified}'")
    print(f"SHA-256 Hash (modified): {sha256_modified_result}")
```

Input:

```
Input String: "Hello Blockchain World!"
```

Expected Output:

```
Original String: 'Hello Blockchain World!'
```

```
MD5 Hash:        3233c09f30e0a5c4048995349e5d483c
SHA-256 Hash:
11a51271676646b9a81f33f669046c075727914e9f73634a36f565507186178e

Original String (modified): 'Hello Blockchain World.'
SHA-256 Hash (modified):
7a1a2b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a4b5c6d7e8f9a
(Note: The SHA-256 hash for the modified string will be completely different,
demonstrating the avalanche effect.)
```

Lab 4: Implement a Digital Signature Algorithm

Title: Lab 4: Implementing a Digital Signature Algorithm

Aim: To understand the concept of digital signatures and implement a simplified version to demonstrate message authenticity and integrity.

Procedure:

1. **Understand Digital Signatures:** Learn how digital signatures use asymmetric cryptography and hashing to verify the sender's identity and ensure the message hasn't been tampered with.
2. **Key Pair Generation (Conceptual):** Understand that a sender needs a public/private key pair.
3. **Hashing the Message:** The sender computes a hash of the message.
4. **Signing the Hash:** The sender encrypts (signs) the message hash using their *private* key. This encrypted hash is the digital signature.
5. **Verification:** The receiver decrypts the signature using the sender's *public* key to get the original hash. The receiver also computes a hash of the received message. If both hashes match, the signature is valid.

Source Code (Conceptual C-like Pseudocode):

```
// This is conceptual pseudocode to illustrate the steps of a digital
signature.
// It does not use actual cryptographic libraries or secure implementations.

// Assume we have functions for:
// - generateKeyPair(): generates a public and private key
// - hash(message): computes a cryptographic hash of the message
// - encryptWithPrivateKey(data, privateKey): encrypts data with a private
key (signing)
// - decryptWithPublicKey(data, publicKey): decrypts data with a public key
(verification)

typedef struct {
    char* message;
    char* signature; // The signed hash
    char* publicKey; // Sender's public key for verification
} SignedMessage;

// Sender's side
SignedMessage createDigitalSignature(char* message, char* privateKey, char*
publicKey) {
    char* messageHash = hash(message); // Step 1: Hash the message
    char* signature = encryptWithPrivateKey(messageHash, privateKey); // Step
2: Sign the hash with private key

    SignedMessage signedMsg;
    signedMsg.message = message;
    signedMsg.signature = signature;
    signedMsg.publicKey = publicKey; // Include public key for receiver
    return signedMsg;
}

// Receiver's side
int verifyDigitalSignature(SignedMessage signedMsg) {
    char* receivedMessageHash = hash(signedMsg.message); // Step 3: Hash the
received message
    char* decryptedHash = decryptWithPublicKey(signedMsg.signature,
signedMsg.publicKey); // Step 4: Decrypt signature with public key
```

```
    // Step 5: Compare the hashes
    if (strcmp(receivedMessageHash, decryptedHash) == 0) {
        return 1; // Signature is valid
    } else {
        return 0; // Signature is invalid (message tampered or wrong sender)
    }
}

// Main conceptual flow
int main() {
    // Conceptual Key Generation
    char* senderPrivateKey = "SENDER_PRIVATE_KEY";
    char* senderPublicKey = "SENDER_PUBLIC_KEY";

    char* originalMessage = "This is a secret message.";

    // Sender creates the signed message
    SignedMessage mySignedMessage = createDigitalSignature(originalMessage,
senderPrivateKey, senderPublicKey);

    printf("Original Message: %s\n", mySignedMessage.message);
    printf("Digital Signature (conceptual): %s\n",
mySignedMessage.signature);

    // Receiver verifies the signed message
    if (verifyDigitalSignature(mySignedMessage)) {
        printf("Verification Result: Signature is VALID. Message is authentic
and untampered.\n");
    } else {
        printf("Verification Result: Signature is INVALID. Message might be
tampered or sender is not authentic.\n");
    }

    // Simulate tampering
    printf("\nSimulating message tampering...\n");
    mySignedMessage.message = "This is a tampered message."; // Message
changed!

    if (verifyDigitalSignature(mySignedMessage)) {
        printf("Verification Result: Signature is VALID. (ERROR: Should be
invalid!)\n");
    } else {
        printf("Verification Result: Signature is INVALID. (Correctly
detected tampering!)\n");
    }

    return 0;
}
```

Input:

```
Original Message: "This is a secret message."
Sender's Private Key: (conceptually generated)
Sender's Public Key: (conceptually generated)
```

Expected Output:

```
Original Message: This is a secret message.
Digital Signature (conceptual): [Some representation of the signed hash]
Verification Result: Signature is VALID. Message is authentic and untampered.

Simulating message tampering...
```

```
Verification Result: Signature is INVALID. (Correctly detected tampering!)
```

Lab 5: Demonstrate the Working of the Merkle Tree Using Any Programming Language

Title: Lab 5: Demonstrating the Working of a Merkle Tree

Aim: To understand the structure and functionality of a Merkle tree (hash tree) and implement a basic version to verify data integrity efficiently.

Procedure:

1. **Understand Merkle Trees:** Learn about how Merkle trees organize hashes of data in a hierarchical structure, allowing for efficient verification of data integrity without downloading all data.
2. **Define Data Blocks:** Start with a list of data blocks (e.g., transactions in a blockchain).
3. **Hash Leaf Nodes:** Compute the hash of each individual data block to form the leaf nodes of the tree.
4. **Build Tree Upwards:** Recursively combine adjacent hashes, hash their concatenation, and move up the tree until a single root hash (Merkle Root) is obtained.
5. **Implement Verification (Conceptual):** Understand how to verify if a specific data block is part of the tree by checking its hash against the Merkle Root using a "Merkle proof."

Source Code (Python):

```
import hashlib

def sha256(data):
    """Helper function to compute SHA-256 hash."""
    return hashlib.sha256(data.encode('utf-8')).hexdigest()

def build_merkle_tree(data_blocks):
    """
    Builds a Merkle tree from a list of data blocks.
    Returns the Merkle root and the tree structure (for demonstration).
    """
    if not data_blocks:
        return None, {}

    # Step 1: Hash leaf nodes
    leaves = [sha256(block) for block in data_blocks]
    tree = {0: leaves} # Store levels of the tree

    current_level = leaves
    level_num = 1

    # Step 2: Build tree upwards
    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            hash1 = current_level[i]
            hash2 = current_level[i+1] if i+1 < len(current_level) else hash1
# Duplicate last hash if odd number
            combined_hash = sha256(hash1 + hash2)
            next_level.append(combined_hash)
        current_level = next_level
        tree[level_num] = next_level
        level_num += 1

    merkle_root = current_level[0] if current_level else None
    return merkle_root, tree
```

```
# Example Usage
if __name__ == "__main__":
    transactions = [
        "Alice sends 10 BTC to Bob",
        "Charlie sends 5 BTC to David",
        "Eve sends 2 BTC to Frank",
        "Grace sends 7 BTC to Harry"
    ]

    print("Data Blocks (Transactions):")
    for i, tx in enumerate(transactions):
        print(f"Tx {i+1}: {tx}")

    merkle_root, merkle_tree_structure = build_merkle_tree(transactions)

    print("\nMerkle Tree Structure (Hashes at each level):")
    for level, hashes in merkle_tree_structure.items():
        print(f"Level {level}: {hashes}")

    print(f"\nMerkle Root: {merkle_root}")

    # Demonstrate verification (conceptual - a full proof would involve path)
    # If any transaction changes, the Merkle Root will change.
    tampered_transactions = [
        "Alice sends 10 BTC to Bob",
        "Charlie sends 5 BTC to David",
        "Eve sends 2 BTC to Frank",
        "Grace sends 8 BTC to Harry" # Tampered!
    ]
    tampered_merkle_root, _ = build_merkle_tree(tampered_transactions)
    print(f"\nMerkle Root with Tampered Transaction: {tampered_merkle_root}")

    if merkle_root == tampered_merkle_root:
        print("Verification: Merkle Root matches (ERROR: should not match if
tampered!)")
    else:
        print("Verification: Merkle Root does NOT match. Data has been
tampered with.")
```

Input:

```
List of data blocks (transactions):
["Alice sends 10 BTC to Bob",
 "Charlie sends 5 BTC to David",
 "Eve sends 2 BTC to Frank",
 "Grace sends 7 BTC to Harry"]
```

Expected Output:

```
Data Blocks (Transactions):
Tx 1: Alice sends 10 BTC to Bob
Tx 2: Charlie sends 5 BTC to David
Tx 3: Eve sends 2 BTC to Frank
Tx 4: Grace sends 7 BTC to Harry

Merkle Tree Structure (Hashes at each level):
Level 0: ['[hash_of_tx1]', '[hash_of_tx2]', '[hash_of_tx3]', '[hash_of_tx4]']
Level 1: ['[hash_of_tx1+tx2]', '[hash_of_tx3+tx4]']
Level 2: ['[hash_of_level1_hash1+level1_hash2]']

Merkle Root: [a_unique_merkle_root_hash]
```

```
Merkle Root with Tampered Transaction: [a_different_merkle_root_hash]
Verification: Merkle Root does NOT match. Data has been tampered with.
```

(Note: Actual hash values will be long hexadecimal strings.)

Lab 7: Study Assignment on Blockchain-Based Applications/Projects

Title: Lab 7: Study Assignment on Blockchain-Based Applications/Projects

Aim: To research and analyze various real-world applications and projects that leverage blockchain technology across different industries.

Procedure:

1. **Identify Key Sectors:** Research and identify industries or domains where blockchain technology is being applied (e.g., finance, supply chain, healthcare, gaming, identity management).
2. **Select Specific Projects:** Choose at least 3-5 distinct blockchain-based applications or projects for detailed study. Examples could include:
   - Decentralized Finance (DeFi) protocols (e.g., Uniswap, Aave)
   - Supply Chain Traceability (e.g., IBM Food Trust, VeChain)
   - Non-Fungible Tokens (NFTs) and their platforms (e.g., OpenSea, CryptoPunks)
   - Decentralized Autonomous Organizations (DAOs)
   - Blockchain in Healthcare (e.g., patient data management)
3. **Analyze Each Project:** For each selected project, investigate and document the following:
   - **Purpose/Problem Solved:** What specific problem does it address?
   - **Blockchain Used:** Which blockchain platform (e.g., Ethereum, Solana, Hyperledger Fabric) is it built on?
   - **Key Features:** What are its primary functionalities and innovations?
   - **Benefits:** What advantages does blockchain bring to this application compared to traditional systems?
   - **Challenges/Limitations:** What are the current obstacles or drawbacks?
   - **Impact:** What is its potential or actual impact on the respective industry?
4. **Synthesize Findings:** Compare and contrast the different applications, identifying common themes, challenges, and future trends in blockchain adoption.
5. **Prepare a Report:** Compile your findings into a structured report.

Source Code: N/A (This is a research and analysis assignment, not a coding task.)

Input:

- Access to internet for research.
- Relevant academic papers, industry reports, project whitepapers, and reputable news articles on blockchain applications.

Expected Output: A comprehensive report (e.g., 5-10 pages) detailing the analysis of selected blockchain-based applications/projects, including:

- An introduction to blockchain applications.
- Detailed sections for each chosen project, covering its purpose, technology, features, benefits, challenges, and impact.
- A comparative analysis of the studied projects.
- A conclusion summarizing key insights and future outlook.

Lab 8: Write a Program to Study Blockchain Using Python

Title: Lab 8: Program to Study Blockchain Using Python

Aim: To implement a simplified blockchain from scratch in Python to understand its core components: blocks, chaining, hashing, and basic proof-of-work.

Procedure:

1. **Define a Block Structure:** Create a `Block` class that includes attributes like index, timestamp, data (transactions), previous hash, nonce, and its own hash.
2. **Calculate Block Hash:** Implement a method within the `Block` class to calculate its hash using SHA-256, incorporating all its attributes.
3. **Implement Proof-of-Work:** Create a simple proof-of-work mechanism (e.g., finding a hash that starts with a certain number of leading zeros) to simulate mining.
4. **Create a Blockchain Class:** Develop a `Blockchain` class to manage the chain of blocks.
   - **Genesis Block:** Implement a method to create the first block (genesis block).
   - **Add New Blocks:** Implement a method to add new blocks to the chain, ensuring they are properly linked to the previous block's hash.
   - **Mining Function:** Integrate the proof-of-work into the block creation process.
   - **Chain Validation (Optional but Recommended):** Add a method to verify the integrity of the entire chain by re-calculating hashes and checking links.

Source Code (Python):

```
import hashlib
import time
import json

class Block:
    def __init__(self, index, timestamp, data, previous_hash, nonce=0):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.previous_hash = previous_hash
        self.nonce = nonce # Used for Proof-of-Work
        self.hash = self.calculate_hash()

    def calculate_hash(self):
        """Calculates the SHA-256 hash of the block's contents."""
        block_string = json.dumps({
            "index": self.index,
            "timestamp": str(self.timestamp),
            "data": self.data,
            "previous_hash": self.previous_hash,
            "nonce": self.nonce
        }, sort_keys=True).encode()
        return hashlib.sha256(block_string).hexdigest()

class Blockchain:
    def __init__(self):
        self.chain = []
        self.difficulty = 2 # Number of leading zeros required for proof-of-
work
        self.create_genesis_block()

    def create_genesis_block(self):
        """Creates the first block in the blockchain."""
        self.add_block(Block(0, time.time(), "Genesis Block", "0"))
```

```
    def get_latest_block(self):
        """Returns the last block in the chain."""
        return self.chain[-1]

    def add_block(self, new_block):
        """Adds a new block to the chain (after mining)."""
        new_block.previous_hash = self.get_latest_block().hash
        new_block.hash = new_block.calculate_hash() # Recalculate hash with
previous_hash set
        self.chain.append(new_block)

    def mine_block(self, data):
        """Mines a new block by finding a valid nonce for the proof-of-
work."""
        latest_block = self.get_latest_block()
        new_block = Block(latest_block.index + 1, time.time(), data,
latest_block.hash)

        print(f"Mining block {new_block.index}...")
        while new_block.hash[:self.difficulty] != '0' * self.difficulty:
            new_block.nonce += 1
            new_block.hash = new_block.calculate_hash()
        print(f"Block {new_block.index} mined: {new_block.hash}")
        self.chain.append(new_block)

    def is_chain_valid(self):
        """Checks if the entire blockchain is valid."""
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i-1]

            # Check if current block's hash is correct
            if current_block.hash != current_block.calculate_hash():
                print(f"Block {current_block.index} hash mismatch!")
                return False

            # Check if current block points to the correct previous hash
            if current_block.previous_hash != previous_block.hash:
                print(f"Block {current_block.index} previous hash mismatch!")
                return False

            # Check proof-of-work
            if current_block.hash[:self.difficulty] != '0' * self.difficulty:
                print(f"Block {current_block.index} does not meet difficulty
requirement!")
                return False
        return True

# Example Usage
if __name__ == "__main__":
    my_blockchain = Blockchain()

    print("Creating Blockchain...")
    print(f"Genesis Block Hash: {my_blockchain.chain[0].hash}")

    my_blockchain.mine_block({"amount": 10, "sender": "Alice", "receiver":
"Bob"})
    my_blockchain.mine_block({"amount": 5, "sender": "Bob", "receiver":
"Charlie"})
    my_blockchain.mine_block({"amount": 20, "sender": "Charlie", "receiver":
"Alice"})

    print("\nBlockchain Status:")
    for block in my_blockchain.chain:
        print(f"Block #{block.index}")
        print(f"  Timestamp: {time.ctime(block.timestamp)}")
```

```
        print(f"  Data: {block.data}")
        print(f"  Previous Hash: {block.previous_hash}")
        print(f"  Nonce: {block.nonce}")
        print(f"  Hash: {block.hash}\n")

    print(f"Is blockchain valid? {my_blockchain.is_chain_valid()}")

    # Demonstrate tampering
    print("\nAttempting to tamper with a block...")
    my_blockchain.chain[1].data = {"amount": 999, "sender": "Tamper",
"receiver": "Evil"}
    my_blockchain.chain[1].hash = my_blockchain.chain[1].calculate_hash() #
Must re-calculate hash of tampered block

    print(f"Is blockchain valid after tampering?
{my_blockchain.is_chain_valid()}")
```

Input:

```
No direct input required, transactions are hardcoded for demonstration.
The program will simulate mining new blocks with example data.
```

Expected Output:

```
Creating Blockchain...
Genesis Block Hash: [hash_value]
Mining block 1...
Block 1 mined: [hash_value_starting_with_00]
Mining block 2...
Block 2 mined: [hash_value_starting_with_00]
Mining block 3...
Block 3 mined: [hash_value_starting_with_00]

Blockchain Status:
Block #0
  Timestamp: [timestamp_of_genesis]
  Data: Genesis Block
  Previous Hash: 0
  Nonce: 0
  Hash: [genesis_block_hash]

Block #1
  Timestamp: [timestamp_of_block1]
  Data: {'amount': 10, 'sender': 'Alice', 'receiver': 'Bob'}
  Previous Hash: [genesis_block_hash]
  Nonce: [nonce_value]
  Hash: [block1_hash]

Block #2
  Timestamp: [timestamp_of_block2]
  Data: {'amount': 5, 'sender': 'Bob', 'receiver': 'Charlie'}
  Previous Hash: [block1_hash]
  Nonce: [nonce_value]
  Hash: [block2_hash]

Block #3
  Timestamp: [timestamp_of_block3]
  Data: {'amount': 20, 'sender': 'Charlie', 'receiver': 'Alice'}
  Previous Hash: [block2_hash]
  Nonce: [nonce_value]
  Hash: [block3_hash]

Is blockchain valid? True
```

```
Attempting to tamper with a block...
Block 2 previous hash mismatch!
Is blockchain valid after tampering? False
```

(Note: Actual hash values and timestamps will vary.)

Lab 9: Case Study on Blockchain Decentralization

Title: Lab 9: Case Study on Blockchain Decentralization

Aim: To conduct a case study on the concept of decentralization within blockchain technology, exploring its importance, different forms, and the challenges associated with achieving and maintaining it.

Procedure:

1. **Define Decentralization:** Clearly define what decentralization means in the context of blockchain, contrasting it with centralized and distributed systems.
2. **Identify Pillars of Decentralization:** Research the key aspects of decentralization in blockchain, including:
    - **Decentralized Governance:** How decisions are made (e.g., through consensus mechanisms, DAOs).
    - **Decentralized Network:** How nodes communicate and validate transactions (e.g., peer-to-peer).
    - **Decentralized Data Storage:** How data is stored across multiple nodes.
    - **Decentralized Development:** How the protocol is maintained and evolved.
3. **Analyze Case Studies:** Choose at least two prominent blockchain networks (e.g., Bitcoin, Ethereum, Solana, Polkadot) and analyze their approach to decentralization:
    - **Bitcoin:** Focus on its highly decentralized nature, proof-of-work, and governance.
    - **Ethereum:** Discuss its evolution, transition to Proof-of-Stake (PoS), and the trade-offs between decentralization, scalability, and security.
    - **Other Blockchains:** Briefly compare with other chains that might prioritize scalability or specific use cases, and how that impacts their decentralization.
4. **Discuss Trade-offs:** Explore the inherent trade-offs in blockchain design, particularly the "Blockchain Trilemma" (decentralization, security, scalability) and how different projects address it.
5. **Challenges to Decentralization:** Identify and discuss potential threats or challenges to decentralization (e.g., mining pool centralization, validator centralization in PoS, regulatory pressures).
6. **Prepare a Report:** Compile your findings into a structured case study report.

Source Code: N/A (This is a research and analysis assignment, not a coding task.)

Input:

- Access to internet for research.
- Whitepapers of major blockchain projects (Bitcoin, Ethereum).
- Academic papers and articles discussing blockchain decentralization, governance, and the blockchain trilemma.

Expected Output: A detailed case study report (e.g., 6-12 pages) on blockchain decentralization, including:

- An introduction to decentralization in blockchain.
- Sections analyzing Bitcoin and Ethereum's decentralization models.
- A discussion on the trade-offs and challenges faced in achieving true decentralization.
- A conclusion summarizing the importance of decentralization and future trends.

Lab 10: Creating Bitcoins (Conceptual Implementation)

Title: Lab 10: Creating Bitcoins (Conceptual Implementation)

Aim: To understand the conceptual process of "creating" new bitcoins through mining, specifically focusing on the block reward mechanism within a simplified blockchain model.

Procedure:

1. **Review Blockchain Basics (from Lab 8):** Revisit the concepts of blocks, chaining, and hashing.
2. **Understand Mining Reward:** Learn how new cryptocurrency units (like bitcoins) are introduced into circulation as a reward to the miner who successfully adds a new block to the blockchain.
3. **Integrate Coinbase Transaction:** Modify the `mine_block` function from Lab 8 to include a "coinbase" transaction. This is a special transaction where the miner creates new coins for themselves as a reward.
4. **Simulate Difficulty Adjustment (Optional):** Briefly discuss how the mining difficulty adjusts over time to maintain a consistent block time.
5. **Observe Coin Creation:** Run the simulation and observe how new "bitcoins" (represented by a reward) are generated with each successfully mined block.

Source Code (Python - Extension of Lab 8):

```
import hashlib
import time
import json

class Block:
    def __init__(self, index, timestamp, data, previous_hash, nonce=0):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.previous_hash = previous_hash
        self.nonce = nonce
        self.hash = self.calculate_hash()

    def calculate_hash(self):
        block_string = json.dumps({
            "index": self.index,
            "timestamp": str(self.timestamp),
            "data": self.data,
            "previous_hash": self.previous_hash,
            "nonce": self.nonce
        }, sort_keys=True).encode()
        return hashlib.sha256(block_string).hexdigest()

class Blockchain:
    def __init__(self):
        self.chain = []
        self.difficulty = 2
        self.mining_reward = 50 # Conceptual reward for mining a block
        self.create_genesis_block()

    def create_genesis_block(self):
        self.add_block(Block(0, time.time(), {"transactions": [], "message":
"Genesis Block"}, "0"))

    def get_latest_block(self):
        return self.chain[-1]
```

```
    def add_block(self, new_block):
        new_block.previous_hash = self.get_latest_block().hash
        new_block.hash = new_block.calculate_hash()
        self.chain.append(new_block)

    def mine_block(self, transactions, miner_address):
        """
        Mines a new block, including a coinbase transaction for the miner.
        """
        # Create a coinbase transaction (miner reward)
        coinbase_tx = {
            "sender": "network", # Special sender for newly created coins
            "receiver": miner_address,
            "amount": self.mining_reward,
            "type": "coinbase"
        }

        # Add coinbase transaction to the beginning of the block's
transactions
        block_data = {"transactions": [coinbase_tx] + transactions}

        latest_block = self.get_latest_block()
        new_block = Block(latest_block.index + 1, time.time(), block_data,
latest_block.hash)

        print(f"Mining block {new_block.index} for {miner_address}...")
        while new_block.hash[:self.difficulty] != '0' * self.difficulty:
            new_block.nonce += 1
            new_block.hash = new_block.calculate_hash()
        print(f"Block {new_block.index} mined: {new_block.hash}")
        self.chain.append(new_block)

# Example Usage
if __name__ == "__main__":
    my_blockchain = Blockchain()
    miner_address_alice = "Alice_Wallet_Address"
    miner_address_bob = "Bob_Wallet_Address"

    print("Creating Blockchain...")
    print(f"Genesis Block Hash: {my_blockchain.chain[0].hash}")

    # Mine first block with some transactions and Alice as miner
    my_blockchain.mine_block(
        [{"amount": 10, "sender": "UserA", "receiver": "UserB"}],
        miner_address_alice
    )

    # Mine second block with some transactions and Bob as miner
    my_blockchain.mine_block(
        [{"amount": 5, "sender": "UserC", "receiver": "UserD"}],
        miner_address_bob
    )

    print("\nBlockchain Status (showing coinbase transactions):")
    for block in my_blockchain.chain:
        print(f"Block #{block.index}")
        print(f"  Timestamp: {time.ctime(block.timestamp)}")
        print(f"  Data: {block.data}")
        print(f"  Previous Hash: {block.previous_hash}")
        print(f"  Nonce: {block.nonce}")
        print(f"  Hash: {block.hash}\n")
```

Input:

```
No direct input. The program simulates transactions and mining.
Miner addresses are hardcoded for demonstration.
```

Expected Output:

```
Creating Blockchain...
Genesis Block Hash: [genesis_hash]
Mining block 1 for Alice_Wallet_Address...
Block 1 mined: [block1_hash_starting_with_00]
Mining block 2 for Bob_Wallet_Address...
Block 2 mined: [block2_hash_starting_with_00]

Blockchain Status (showing coinbase transactions):
Block #0
  Timestamp: [timestamp]
  Data: {'transactions': [], 'message': 'Genesis Block'}
  Previous Hash: 0
  Nonce: 0
  Hash: [genesis_hash]

Block #1
  Timestamp: [timestamp]
  Data: {'transactions': [{'sender': 'network', 'receiver':
'Alice_Wallet_Address', 'amount': 50, 'type': 'coinbase'}], 'message': 'UserA
sends 10 to UserB'}
  Previous Hash: [genesis_hash]
  Nonce: [nonce]
  Hash: [block1_hash]

Block #2
  Timestamp: [timestamp]
  Data: {'transactions': [{'sender': 'network', 'receiver':
'Bob_Wallet_Address', 'amount': 50, 'type': 'coinbase'}], 'message': 'UserC
sends 5 to UserD'}
  Previous Hash: [block1_hash]
  Nonce: [nonce]
  Hash: [block2_hash]
```

(Note: Actual hash values and timestamps will vary. The `message` in `block_data` for mined blocks might need adjustment to correctly reflect the combined data.)

Lab 11: Case Study for Bitcoin Generation Mechanisms

Title: Lab 11: Case Study for Bitcoin Generation Mechanisms

Aim: To conduct a detailed case study on how new bitcoins are generated and introduced into circulation, including the role of mining, block rewards, and the halving events.

Procedure:

1. **Understand Bitcoin Mining:** Research the process of Bitcoin mining, focusing on the computational puzzle (Proof-of-Work) and its purpose in securing the network.
2. **Block Reward Mechanism:** Investigate the concept of the "block reward" – the amount of new bitcoins granted to the miner who successfully finds a new block.
3. **Bitcoin Halving Events:** Study the halving mechanism, where the block reward is periodically cut in half (approximately every four years or 210,000 blocks). Understand its impact on supply and scarcity.
4. **Total Supply Limit:** Research the fixed total supply of Bitcoin (21 million coins) and how the halving mechanism ensures this limit is gradually approached.
5. **Transaction Fees:** Understand how transaction fees also contribute to a miner's earnings, especially as block rewards decrease over time.
6. **Economic Implications:** Discuss the economic implications of these generation mechanisms, such as their effect on inflation, scarcity, and miner incentives.
7. **Prepare a Report:** Compile your findings into a structured case study report.

Source Code: N/A (This is a research and analysis assignment, not a coding task.)

Input:

- Access to internet for research.
- The original Bitcoin Whitepaper by Satoshi Nakamoto.
- Reliable sources on Bitcoin economics, mining, and halving events (e.g., CoinDesk, Investopedia, academic papers).
- Historical data on Bitcoin block rewards and halving dates.

Expected Output: A comprehensive case study report (e.g., 8-15 pages) on Bitcoin generation mechanisms, including:

- An introduction to Bitcoin's monetary policy.
- Detailed explanations of Proof-of-Work and the block reward.
- An in-depth analysis of Bitcoin halving events, their history, and future projections.
- Discussion on the total supply limit and its significance.
- The role of transaction fees.
- Economic analysis of Bitcoin's supply dynamics.
- A conclusion summarizing the sustainability and implications of Bitcoin's generation model.

Lab 12: Building a Bitcoin Wallet Application Using Any Programming Languages/Tools (Conceptual)

Title: Lab 12: Building a Bitcoin Wallet Application (Conceptual)

Aim: To understand the fundamental components and conceptual design of a cryptocurrency wallet, focusing on key generation, address creation, and transaction signing.

Procedure:

1. **Understand Wallet Types:** Research different types of cryptocurrency wallets (e.g., hot vs. cold, software vs. hardware, custodial vs. non-custodial).
2. **Key Pair Generation:** Learn how a public-private key pair is generated using cryptographic algorithms (e.g., ECDSA for Bitcoin).
3. **Address Generation:** Understand how a Bitcoin address is derived from the public key (involving hashing and encoding).
4. **Transaction Structure (Conceptual):** Grasp the basic structure of a Bitcoin transaction (inputs, outputs, amounts).
5. **Transaction Signing:** Learn how the private key is used to digitally sign a transaction, proving ownership of the funds without revealing the private key.
6. **Broadcasting Transaction (Conceptual):** Understand that a signed transaction is then broadcasted to the Bitcoin network for validation and inclusion in a block.
7. **Conceptual Implementation:** Provide a simplified code example focusing on key generation and address derivation.

Source Code (Python - Conceptual Key & Address Generation):

```
# This is a highly simplified conceptual example.
# It does NOT use actual Bitcoin-specific cryptographic libraries or secure
practices.
# Real Bitcoin wallets use complex ECDSA curve secp256k1, hashing, and
encoding.

import hashlib
import random

def generate_private_key_conceptual():
    """Generates a conceptual 'private key' (a large random number)."""
    # In reality, this is a 256-bit integer.
    return hex(random.getrandbits(256))[2:] # Convert to hex string

def derive_public_key_conceptual(private_key):
    """Conceptually derives a 'public key' from a private key."""
    # In reality, this involves elliptic curve multiplication.
    # Here, we'll just hash the private key for a conceptual public key.
    return hashlib.sha256(private_key.encode('utf-8')).hexdigest()

def derive_bitcoin_address_conceptual(public_key):
    """Conceptually derives a 'Bitcoin address' from a public key."""
    # In reality, this involves multiple hashing steps (SHA256, RIPEMD160)
    # and Base58Check encoding.
    # Here, we'll just take a slice of the public key hash for simplicity.
    return "1" + public_key[:30] # Bitcoin addresses typically start with '1'
or 'bc1'

# Example Usage
if __name__ == "__main__":
    print("Generating a conceptual Bitcoin wallet key pair and address:")
```

```
    private_key = generate_private_key_conceptual()
    public_key = derive_public_key_conceptual(private_key)
    bitcoin_address = derive_bitcoin_address_conceptual(public_key)

    print(f"Conceptual Private Key: {private_key}")
    print(f"Conceptual Public Key:  {public_key}")
    print(f"Conceptual Bitcoin Address: {bitcoin_address}")

    print("\nImportant: In a real wallet, these operations are
cryptographically secure and complex.")
```

Input:

```
No direct input. The program generates keys and address.
```

Expected Output:

```
Generating a conceptual Bitcoin wallet key pair and address:
Conceptual Private Key: [a_long_hex_string_representing_private_key]
Conceptual Public Key:  [a_long_hex_string_representing_public_key_hash]
Conceptual Bitcoin Address: 1[first_30_chars_of_public_key_hash]

Important: In a real wallet, these operations are cryptographically secure
and complex.
```

(Note: Actual key and address values will be random and different each time.)

Lab 13: Creating a Cryptocurrency Wallet Using Java (Conceptual)

Title: Lab 13: Creating a Cryptocurrency Wallet Using Java (Conceptual)

Aim: To design and conceptually implement a basic cryptocurrency wallet in Java, focusing on the generation of cryptographic key pairs.

Procedure:

1. **Understand Java Cryptography Architecture (JCA):** Familiarize yourself with Java's built-in cryptography capabilities, particularly `KeyPairGenerator` and `Signature` classes.
2. **Key Pair Generation:** Implement code to generate an asymmetric key pair (e.g., using RSA or ECDSA, though for actual crypto, ECDSA with `secp256k1` curve is needed for Bitcoin-like wallets).
3. **Store Keys (Conceptual):** Understand how private keys would be securely stored (though not implemented in this basic example).
4. **Conceptual Address Derivation:** Briefly discuss how a public key would be transformed into a cryptocurrency address.
5. **Conceptual Transaction Signing:** Understand how the private key would be used to sign a transaction.

Source Code (Java - Conceptual Key Pair Generation):

```
// This is a highly simplified conceptual example.
// It does NOT implement actual Bitcoin-specific cryptography (e.g.,
secp256k1 curve).
// For real cryptocurrency wallets, specialized libraries are required.

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.util.Base64; // For encoding keys to strings

public class ConceptualCryptoWallet {

    public static void main(String[] args) {
        try {
            // Step 1: Initialize KeyPairGenerator
            // For a real Bitcoin-like wallet, you'd use "EC" (Elliptic
Curve)
            // and specify the "secp256k1" curve parameters.
            // Here, we use "RSA" for simplicity as it's commonly available.
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
            keyGen.initialize(2048); // Key size

            // Step 2: Generate KeyPair
            KeyPair pair = keyGen.generateKeyPair();
            PrivateKey privateKey = pair.getPrivate();
            PublicKey publicKey = pair.getPublic();

            System.out.println("Conceptual Cryptocurrency Wallet Key Pair
Generation:");
            System.out.println("---------------------------------------------
------");

            // Step 3: Display Keys (encoded for readability)
            System.out.println("Conceptual Private Key (Base64 encoded):");
```

```
System.out.println(Base64.getEncoder().encodeToString(privateKey.getEncoded()
));

            System.out.println("\nConceptual Public Key (Base64 encoded):");

System.out.println(Base64.getEncoder().encodeToString(publicKey.getEncoded())
);

            System.out.println("\nNote: In a real wallet, public key is
further hashed and encoded to form an address.");
            System.out.println("Private key must be kept absolutely secret
and secure.");

        } catch (NoSuchAlgorithmException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

Input:

```
No direct input. The program generates keys.
```

Expected Output:

```
Conceptual Cryptocurrency Wallet Key Pair Generation:
---------------------------------------------------
Conceptual Private Key (Base64 encoded):
[A_long_Base64_encoded_string_representing_the_private_key]

Conceptual Public Key (Base64 encoded):
[A_long_Base64_encoded_string_representing_the_public_key]

Note: In a real wallet, public key is further hashed and encoded to form an
address.
Private key must be kept absolutely secret and secure.
```

(Note: Actual key values will be different each time.)

Lab 14: Code to Implement Peer-to-Peer Using Blockchain

Title: Lab 14: Implementing Peer-to-Peer Communication in a Blockchain Network

Aim: To understand and conceptually implement how nodes in a blockchain network communicate in a peer-to-peer (P2P) fashion to share blocks, transactions, and maintain a synchronized ledger.

Procedure:

1. **Understand P2P Networks:** Learn the basics of peer-to-peer network architecture, where each node can act as both a client and a server.
2. **Node Discovery (Conceptual):** Understand how new nodes discover existing nodes in the network (e.g., through a list of known nodes or DNS seeds).
3. **Message Types:** Identify common messages exchanged in a blockchain P2P network (e.g., `get_blocks`, `send_block`, `get_transactions`, `send_transaction`, `version`, `addr`).
4. **Basic Socket Communication:** Implement a simple client-server model using sockets to simulate two nodes communicating.
5. **Simulate Block/Transaction Propagation:** Demonstrate how a newly mined block or a new transaction would be broadcasted from one node to its peers.
6. **Conceptual Consensus:** Briefly discuss how P2P communication is essential for achieving consensus on the state of the blockchain.

Source Code (Python - Conceptual P2P Simulation using Sockets):

```
# This is a highly simplified conceptual example of P2P communication.
# It does NOT implement a full blockchain P2P network, which is very complex.
# It demonstrates basic socket communication between two simulated "nodes".

import socket
import threading
import time

# --- Node 1 (Server) ---
def node1_server():
    HOST = '127.0.0.1' # Standard loopback interface address (localhost)
    PORT = 65432       # Port to listen on (non-privileged ports are > 1023)

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen()
        print(f"Node 1 (Server) listening on {HOST}:{PORT}")
        conn, addr = s.accept()
        with conn:
            print(f"Node 1 connected by {addr}")
            while True:
                data = conn.recv(1024)
                if not data:
                    break
                message = data.decode('utf-8')
                print(f"Node 1 received: {message}")
                if message == "request_block":
                    response = "Here's Block #123: {'data': 'New 
Transaction'}"
                    conn.sendall(response.encode('utf-8'))
                    print(f"Node 1 sent: {response}")
                elif message == "ping":
                    conn.sendall("pong".encode('utf-8'))
                    print("Node 1 sent: pong")
```

```
                time.sleep(1) # Simulate some processing delay

# --- Node 2 (Client) ---
def node2_client():
    HOST = '127.0.0.1'
    PORT = 65432

    time.sleep(2) # Give server time to start

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        try:
            s.connect((HOST, PORT))
            print(f"Node 2 (Client) connected to {HOST}:{PORT}")

            messages_to_send = ["ping", "request_block", "ping",
"send_transaction: {'amount': 5, 'from': 'B', 'to': 'C'}"]
            for msg in messages_to_send:
                print(f"Node 2 sending: {msg}")
                s.sendall(msg.encode('utf-8'))
                data = s.recv(1024)
                if data:
                    print(f"Node 2 received: {data.decode('utf-8')}")
                time.sleep(3) # Wait for response and next message
        except ConnectionRefusedError:
            print("Node 2: Connection refused. Is Node 1 (server) running?")
        except Exception as e:
            print(f"Node 2 error: {e}")

# Example Usage
if __name__ == "__main__":
    print("Starting conceptual P2P simulation...")

    # Start Node 1 (server) in a separate thread
    server_thread = threading.Thread(target=node1_server)
    server_thread.daemon = True # Allow main program to exit even if thread
is running
    server_thread.start()

    # Start Node 2 (client) in the main thread
    node2_client()

    print("\nConceptual P2P simulation finished.")
```

Input:

```
No direct input. The program simulates communication between two nodes.
```

Expected Output:

```
Starting conceptual P2P simulation...
Node 1 (Server) listening on 127.0.0.1:65432
Node 1 connected by ('127.0.0.1', [some_port])
Node 2 (Client) connected to 127.0.0.1:65432
Node 2 sending: ping
Node 1 received: ping
Node 1 sent: pong
Node 2 received: pong
Node 2 sending: request_block
Node 1 received: request_block
Node 1 sent: Here's Block #123: {'data': 'New Transaction'}
Node 2 received: Here's Block #123: {'data': 'New Transaction'}
Node 2 sending: ping
Node 1 received: ping
```

```
Node 1 sent: pong
Node 2 received: pong
Node 2 sending: send_transaction: {'amount': 5, 'from': 'B', 'to': 'C'}
Node 1 received: send_transaction: {'amount': 5, 'from': 'B', 'to': 'C'}

Conceptual P2P simulation finished.
```

Lab 15: Case Study on Applications of Bitcoins

Title: Lab 15: Case Study on Applications of Bitcoin

Aim: To research and analyze the diverse applications and use cases of Bitcoin beyond its primary role as a digital currency, exploring its impact on various sectors.

Procedure:

1. **Bitcoin as Digital Gold/Store of Value:** Analyze Bitcoin's role as a hedge against inflation and a safe-haven asset, often referred to as "digital gold."
2. **Remittances and Cross-Border Payments:** Investigate how Bitcoin can facilitate faster and cheaper international money transfers, bypassing traditional banking systems.
3. **Censorship Resistance and Financial Freedom:** Discuss Bitcoin's use in regions with unstable economies or restrictive financial controls, providing an alternative for individuals.
4. **Micro-transactions and Programmable Money (Conceptual):** While less common directly on the Bitcoin mainnet due to fees, explore the potential for micro-transactions via layer-2 solutions (like Lightning Network) and the idea of programmable money.
5. **Decentralized Identity/Proof of Existence (Conceptual):** Briefly touch upon how the Bitcoin blockchain can be used to timestamp data or prove the existence of digital assets without relying on a central authority.
6. **Investment and Speculation:** Acknowledge Bitcoin's significant role as an investment asset and a speculative vehicle.
7. **Challenges and Limitations:** Discuss the current challenges in Bitcoin adoption for various applications (e.g., volatility, scalability concerns on layer 1, regulatory uncertainty).
8. **Prepare a Report:** Compile your findings into a structured case study report.

Source Code: N/A (This is a research and analysis assignment, not a coding task.)

Input:

- Access to internet for research.
- Financial news outlets, cryptocurrency analysis platforms, and academic papers discussing Bitcoin's various use cases.
- Reports on global remittance markets and financial inclusion.

Expected Output: A detailed case study report (e.g., 7-14 pages) on the applications of Bitcoin, including:

- An introduction to Bitcoin's multifaceted utility.
- Sections dedicated to each major application area (store of value, remittances, censorship resistance, etc.).
- Analysis of the benefits and challenges of Bitcoin in these applications.
- Discussion on the role of layer-2 solutions.
- A conclusion summarizing Bitcoin's current and potential impact on the global financial landscape.