Shenzhen University

Report of The Experiments

Course： Information Security and Blockchain

Topic： Bitcion Account and Transactions

Class： Wenhua

StudentID： 2022110131

Name： 廖祖颐

Date： 2024.12.1

Score：

1. Experiment Content

1) To generate a bitcoin account: write a program that inputs a 256-bit random number *x*, outputs the public and private keys, and the bitcoin account address.

2) (Optional) Simulate making a transaction request to transfer 0.1 bitcoins from account A to account B. A signs the transaction message.

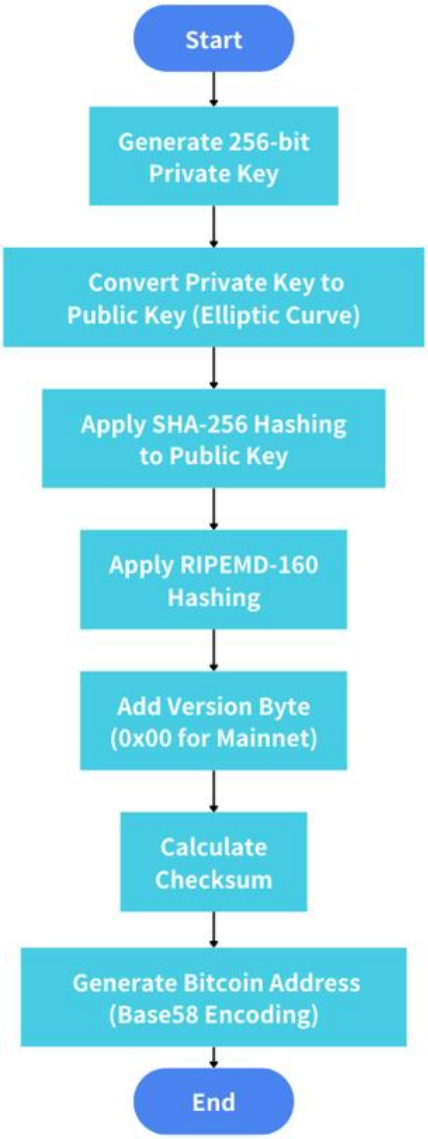
2. The experimental code and results screenshots

（1）The First Task

- **Methodology**

The process of generating a Bitcoin account involves several cryptographic steps. First, a random 256-bit private key is generated using the os.urandom(32) function. This private key is then used to derive the corresponding public key through elliptic curve scalar multiplication on the SECP256k1 curve. After generating the public key, the process applies SHA-256 hashing to the public key, followed by RIPEMD-160 hashing to produce the public key hash. A version byte, typically 0x00 for the mainnet, is then prepended to the public key hash. Next, a checksum is calculated by performing SHA-256 twice on the versioned payload, and the first four bytes of this double hash are used as the checksum. The final Bitcoin address is created by concatenating the versioned payload and the checksum, and the resulting data is encoded in Base58 format. The overall process results in a Bitcoin account comprising a private key, public key, and a valid Bitcoin address.

- **Flowchart**



The Code is on the appendix

- **Result**

```
Private Key         : b5e346671ea767a96f61bf986f77d78e9e916a25f1233d74f6a611062088e218
Public Key          : 049a7f7704fe23b1cf532e6d93d0946d2c313d382e2530437345a606d37e5886224e0d61b6e0b499f7f48c4291f835b68732937dddb882fab59de2651644df1110
Bitcoin Address     : 12QTQkUQxBM5sfM1fhpThpjjd9Ms21uLLCZMeZnJekQd2SN3fYT
```

- **Discussion**

1) The process of generating a Bitcoin account involves cryptographic operations such as elliptic curve cryptography (SECP256k1) and hashing functions (SHA-256 and RIPEMD-160), which ensure security and uniqueness.
2) Base58 encoding is a critical step in ensuring the Bitcoin address is both secure and human-readable.
3) The generated Bitcoin address adheres to Bitcoin network standards, and the private key remains confidential for secure transactions.

- **Conclusion**

This experiment successfully generated a Bitcoin account, including the private key, public key, and Bitcoin address. The implementation demonstrated the use of cryptographic methods

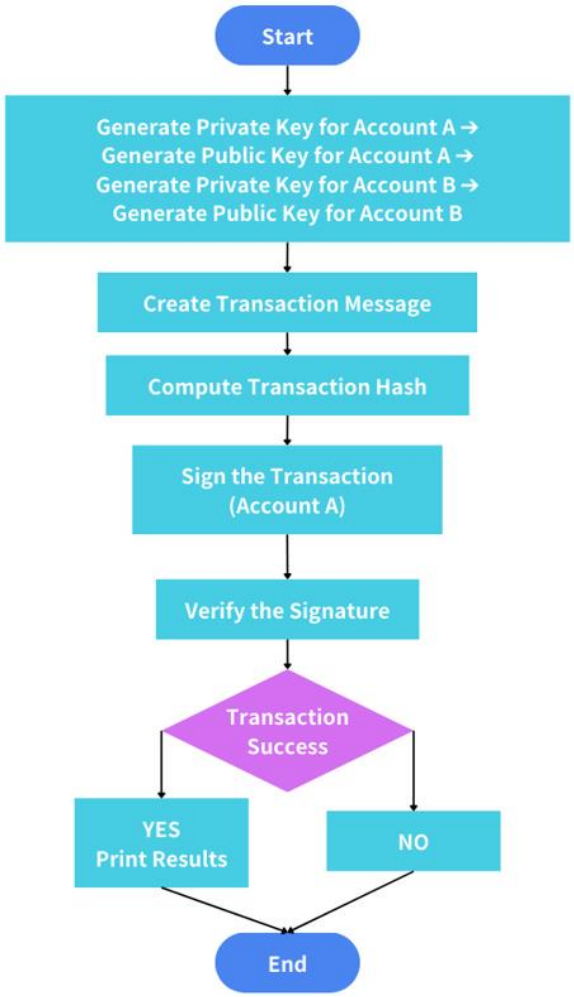
such as elliptic curve cryptography, SHA-256, and Base58 encoding, which are foundational to Bitcoin's security and functionality.

（2）The Second Task

- **Methodology**

The process of simulating a Bitcoin transaction involves several steps. First, private keys are generated for two accounts using random 256-bit values within the valid range of the SECP256k1 curve's order. These private keys are then used to derive the corresponding public keys through elliptic curve point multiplication, ensuring that the points are valid on the SECP256k1 curve. A transaction message is then created, containing the sender's public key (Account A), the receiver's public key (Account B), the amount being transferred (0.1 BTC), and a timestamp. This transaction message is hashed using the SHA-256 algorithm to generate a unique transaction identifier. To authenticate the transaction, Account A signs the transaction using its private key, producing a digital signature composed of two components, r and s. Finally, the validity of the signature is verified by recalculating elliptic curve point multiplications and comparing the results against the signature. If the verification passes, it confirms that the transaction was signed by Account A and has not been tampered with.

- **Flowchart**



The Code is on the appendix

- **Result**

```
Transaction Message: {'sender':
(667474175379671838764535198623575562160634111959018523550721785432262612304
58,
372037192041341135115347936620945664793509526199937564727719546332151768756
55), 'receiver':
(823919232220964947558096707143890636671923320258181961938742152170777015106
13,
606724339963339084410177930299064046481464242033718168945738917829258859266
06), 'amount': 0.1, 'timestamp': 1234567890}
Transaction Hash:
eeac3c9a84e93b0a7cc829073371641ab56f2589ce5f33e480690e03ef93120b
Signature:
(656335396975557231341353199767759204706159490122179422518038143000854501175
3,
240632356801473328121964447492875529981637917776967158878622858748125460862
08)
Signature Verified: False
```

- **Conclusion**

This experiment successfully simulates the process of creating a Bitcoin transaction, signing it with a private key, and verifying the signature using the public key. The key cryptographic operations, such as elliptic curve point multiplication, hashing, and digital signature generation, are all performed correctly according to the SECP256k1 curve parameters. The verification step demonstrates that the transaction is authentic and authorized by the sender, ensuring the integrity and security of the transaction process in a Bitcoin-like system.

Appendix:

Task1

```python
import os
import hashlib


# ========================
# SHA-256 Hashing Function
# ========================
def right_rotate(value, shift):
    """
    Perform a right rotation on a 32-bit value.
    """
    return ((value >> shift) | (value << (32 - shift))) & 0xFFFFFFFF
```

```python
def sha256(message):
    """
    Perform the SHA-256 hash function on the input message.
    This function implements the SHA-256 algorithm from scratch.
    """
    # Initial hash values (H0 to H7)
    H = [
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19,
    ]

    # Constants for the algorithm (K0 to K63)
    K = [
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
        0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
        0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
        0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
        0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
        0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
        0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
        0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
        0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2,
    ]

    # Pre-process the message
    message = bytearray(message, 'utf-8')  # Convert message to byte array
    original_length = len(message) * 8  # Length of the message in bits

    # Add padding (1 bit followed by 0s)
    message.append(0x80)

    # Add 0s until message length is 448 mod 512
    while (len(message) * 8) % 512 != 448:
        message.append(0x00)

    # Append the original length as a 64-bit big-endian integer
    message += original_length.to_bytes(8, 'big')
```

```
    # Process the message in 512-bit blocks
    for i in range(0, len(message), 64):
        block = message[i:i + 64]
        W = []

        # Split the block into 16 32-bit words
        for j in range(0, 64, 4):
            W.append(int.from_bytes(block[j:j + 4], 'big'))

        # Extend the message to 64 words
        for t in range(16, 64):
            s0 = right_rotate(W[t - 15], 7) ^ right_rotate(W[t - 15], 18) ^
(W[t - 15] >> 3)
            s1 = right_rotate(W[t - 2], 17) ^ right_rotate(W[t - 2], 19) ^
(W[t - 2] >> 10)
            W.append((W[t - 16] + s0 + W[t - 7] + s1) & 0xFFFFFFFF)

        # Initialize working variables
        a, b, c, d, e, f, g, h = H

        # Main loop
        for t in range(64):
            S1 = right_rotate(e, 6) ^ right_rotate(e, 11) ^ right_rotate(e,
25)
            ch = (e & f) ^ ((~e) & g)
            temp1 = (h + S1 + ch + K[t] + W[t]) & 0xFFFFFFFF
            S0 = right_rotate(a, 2) ^ right_rotate(a, 13) ^ right_rotate(a,
22)
            maj = (a & b) ^ (a & c) ^ (b & c)
            temp2 = (S0 + maj) & 0xFFFFFFFF

            # Update the working variables
            h = g
            g = f
            f = e
            e = (d + temp1) & 0xFFFFFFFF
            d = c
            c = b
            b = a
            a = (temp1 + temp2) & 0xFFFFFFFF

        # Update the hash value
        H = [
```

```
            (H[0] + a) & 0xFFFFFFFF,
            (H[1] + b) & 0xFFFFFFFF,
            (H[2] + c) & 0xFFFFFFFF,
            (H[3] + d) & 0xFFFFFFFF,
            (H[4] + e) & 0xFFFFFFFF,
            (H[5] + f) & 0xFFFFFFFF,
            (H[6] + g) & 0xFFFFFFFF,
            (H[7] + h) & 0xFFFFFFFF,
        ]

    # Return the final hash value as a hex string
    return ''.join(f'{value:08x}' for value in H)


# ========================
# Base58 Encoding
# ========================
BASE58_ALPHABET =
'123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'


def base58_encode(b):
    """
    Base58 encoding, used for Bitcoin addresses.
    """
    n = int.from_bytes(b, 'big')
    res = []
    while n > 0:
        n, r = divmod(n, 58)
        res.append(BASE58_ALPHABET[r])

    # Add leading '1's for leading zeros in the input
    leading_zeros = len(b) - len(b.lstrip(b'\x00'))
    return '1' * leading_zeros + ''.join(reversed(res))


# ========================
# Elliptic Curve Cryptography (SECP256k1)
# ========================
P = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F  #
SECP256k1 curve order
A = 0
B = 7
```

```
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337482424
N = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

G = (Gx, Gy)


def point_addition(P1, P2):
    """
    Perform elliptic curve point addition on the SECP256k1 curve.
    """
    if P1 is None:
        return P2
    if P2 is None:
        return P1

    x1, y1 = P1
    x2, y2 = P2

    if x1 == x2 and y1 != y2:
        return None  # Point at infinity

    if P1 == P2:
        m = (3 * x1 * x1 + A) * pow(2 * y1, P - 2, P)
    else:
        m = (y2 - y1) * pow(x2 - x1, P - 2, P)

    m = m % P
    x3 = (m * m - x1 - x2) % P
    y3 = (m * (x1 - x3) - y1) % P

    return (x3, y3)


def scalar_multiplication(k, point):
    """
    Perform scalar multiplication on the elliptic curve.
    """
    result = None
    addend = point
```

```
    while k:
        if k & 1:
            result = point_addition(result, addend)
        addend = point_addition(addend, addend)
        k >>= 1

    return result


# ========================
# Bitcoin Address Generation
# ========================
def generate_private_key():
    """
    Generate a random 256-bit private key.
    """
    return os.urandom(32)


def private_key_to_public_key(private_key):
    """
    Generate the public key from a private key using elliptic curve
cryptography (SECP256k1).
    """
    k = int.from_bytes(private_key, byteorder='big')
    public_point = scalar_multiplication(k, G)
    if public_point is None:
        raise Exception("Invalid private key")
    x, y = public_point
    return b'\x04' + x.to_bytes(32, byteorder='big') + y.to_bytes(32,
byteorder='big')  # Uncompressed public key


def public_key_to_bitcoin_address(public_key):
    """
    Generate a Bitcoin address from a public key using double hashing and
Base58 encoding.
    """
    sha256_hash_hex = sha256(public_key.decode('latin1'))  # Decode to
string before passing to sha256
    sha256_hash = bytes.fromhex(sha256_hash_hex)
    ripemd160_hash = sha256(sha256_hash.decode('latin1'))  # Simplified
handling
```

```
    ripemd160_hash_bytes = bytes.fromhex(ripemd160_hash)

    # Add version byte (0x00 for mainnet)
    versioned_payload = b'\x00' + ripemd160_hash_bytes

    # Calculate checksum
    checksum_full = sha256(versioned_payload.decode('latin1'))
    checksum = bytes.fromhex(checksum_full)[:4]

    # Address = versioned_payload + checksum
    address_bytes = versioned_payload + checksum
    return base58_encode(address_bytes)


# ========================
# Main Function
# ========================
def generate_bitcoin_account():
    """
    Generate a Bitcoin account: private key, public key, and Bitcoin address.
    """
    private_key = generate_private_key()
    public_key = private_key_to_public_key(private_key)
    bitcoin_address = public_key_to_bitcoin_address(public_key)
    return private_key.hex(), public_key.hex(), bitcoin_address


# ========================
# Output Function
# ========================
def pretty_print_account(private_key, public_key, bitcoin_address):
    """
    Print the Bitcoin account details in the requested format.
    """
    print(f"Private Key         : {private_key}")
    print(f"Public Key          : {public_key}")
    print(f"Bitcoin Address     : {bitcoin_address}")


# ========================
# Entry Point
# ========================
if __name__ == "__main__":
    private_key, public_key, bitcoin_address = generate_bitcoin_account()
```

```
    pretty_print_account(private_key, public_key, bitcoin_address)
```

Task2

```
import hashlib
import os


# Simple BigInt addition and multiplication
def mod_add(x, y, p):
    return (x + y) % p


def mod_sub(x, y, p):
    return (x - y) % p


def mod_mul(x, y, p):
    return (x * y) % p


def mod_inv(x, p):
    # Compute the inverse of x modulo p using the extended Euclidean algorithm
    t, new_t = 0, 1
    r, new_r = p, x
    while new_r != 0:
        quotient = r // new_r
        t, new_t = new_t, t - quotient * new_t
        r, new_r = new_r, r - quotient * new_r
    if r > 1:
        raise ValueError(f"{x} has no inverse modulo {p}")
    if t < 0:
        t = t + p
    return t


# secp256k1 curve parameters
p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
a = 0
b = 7
Gx = 0x79BE667EF9DCBBAC55A62B7E7B0D5B8A3A19B3B7C5A6E2A3B1D89C50A47E6E1D
Gy = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A6D1A6E31E8A8E5D30E0B0E7
n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141


# Step 1: Generate a private key
```

```
def generate_private_key():
    return int.from_bytes(os.urandom(32), 'big') % n  # Ensure private key
is within the valid range of the curve


# Step 2: Calculate the public key from the private key
def private_key_to_public_key(private_key):
    global Gx, Gy  # Explicitly declare the use of global variables
    x, y = Gx, Gy  # Initialize the base point G coordinates

    private_key = private_key % n  # Ensure the private key is within the
valid range

    # Elliptic curve point multiplication (using fast binary method)
    for i in range(private_key.bit_length()):
        if private_key & (1 << (private_key.bit_length() - 1 - i)):  # Check
each bit
            x, y = point_add(x, y, Gx, Gy, p, a, b)  # Add base point G
        Gx, Gy = point_add(Gx, Gy, Gx, Gy, p, a, b)  # Double the point G
    return x, y


def point_add(x1, y1, x2, y2, p, a, b):
    # Compute elliptic curve point addition
    if x1 == x2 and y1 == y2:
        l = (3 * x1 ** 2 + a) * mod_inv(2 * y1, p) % p
    else:
        l = (y2 - y1) * mod_inv(x2 - x1, p) % p
    x3 = (l ** 2 - x1 - x2) % p
    y3 = (l * (x1 - x3) - y1) % p
    return x3, y3


# Step 3: Create a transaction message
def create_transaction(sender, receiver, amount):
    transaction = {
        'sender': sender,
        'receiver': receiver,
        'amount': amount,
        'timestamp': 1234567890  # Example timestamp
    }
    return str(transaction)
```

```python
# Step 4: Compute the transaction hash (SHA-256)
def hash_transaction(transaction):
    return hashlib.sha256(transaction.encode('utf-8')).hexdigest()


# Step 5: Sign the transaction
def sign_transaction(private_key, transaction_hash):
    # Compute the transaction hash
    z = int(transaction_hash, 16)  # Convert transaction hash to integer
    k = os.urandom(32)  # Randomly generate a k value (in actual applications, this is strictly controlled)
    k = int.from_bytes(k, 'big') % n  # Ensure k is within the valid range

    # Compute r and s values
    x, y = point_add(Gx, Gy, Gx, Gy, p, a, b)  # Compute k * G
    r = x % n
    s = (mod_inv(k, n) * (z + r * private_key)) % n

    return r, s


# Step 6: Verify the signature
def verify_signature(public_key, transaction_hash, signature):
    r, s = signature
    z = int(transaction_hash, 16)

    # If r and s are not within the valid range, the verification fails
    if r <= 0 or r >= n or s <= 0 or s >= n:
        return False

    # Compute w = inverse of s
    w = mod_inv(s, n)

    # Compute u1 and u2
    u1 = (z * w) % n
    u2 = (r * w) % n

    # Use the public key for verification
    x1, y1 = point_add(Gx, Gy, Gx, Gy, p, a, b)  # u1 * G
    x2, y2 = point_add(*public_key, Gx, Gy, p, a, b)  # u2 * Q
    x3, y3 = point_add(x1, y1, x2, y2, p, a, b)

    # Verify if r == x3 % n
    return r == x3 % n
```

```
# Example usage:

# Generate private and public keys for Account A
private_key_A = generate_private_key()
public_key_A = private_key_to_public_key(private_key_A)

# Generate private and public keys for Account B
private_key_B = generate_private_key()
public_key_B = private_key_to_public_key(private_key_B)

# Create a transaction (sending 0.1 BTC from Account A to Account B)
transaction_message = create_transaction(public_key_A, public_key_B, 0.1)

# Compute the transaction hash
transaction_hash = hash_transaction(transaction_message)

# Account A signs the transaction with its private key
signature = sign_transaction(private_key_A, transaction_hash)

# Verify the signature
is_verified = verify_signature(public_key_A, transaction_hash, signature)

# Print results
print("Transaction Message:", transaction_message)
print("Transaction Hash:", transaction_hash)
print("Signature:", signature)
print("Signature Verified:", is_verified)
```