COMP7200 Programming Project
Implementation of a Mini Blockchain

Name	Student Number	Contributions
Zhang Yu	24436100	● Task 4.1-Transaction generation (code and report) ● Introduction of report
Zhou Yingtong	24422231	● Task 4.2-Veriffable Merkle tree of transactions (code and report) ● Conclusion of report
Chan Wai Lam	24475505	● Task 4.3- Construction of blockchain (code and report) ● Document integration
Tai Long Ching	24464163	● Leader of the group ● Task 4.4- Mining a block (code and report) ● Pipeline flow ● README
Wang Shiyi	24442399	● Task 4.5- Integrity Verification ● (code ('is_chain_valid' function in class 'BlockChain' and all of '4.5.1 Integrity Verification') and report), ● Experimental results (output of test codes 4.5.1), ● References of report

1 Introduction

Blockchains are tamper evident and tamper resistant digital ledgers implemented in a distributed fashion (i.e., without a central repository) and usually without a central authority (i.e., a bank, company, or government)[1]. Blockchain has three basic elements, namely transaction, block and chain. Through a decentralized network structure, blockchain technology enables the distributed sharing and management of data, thus ensuring the security and reliability of data without the need to trust intermediaries. Due to these significant advantages, blockchain technology is full of potential in many industries, especially in financial currency.

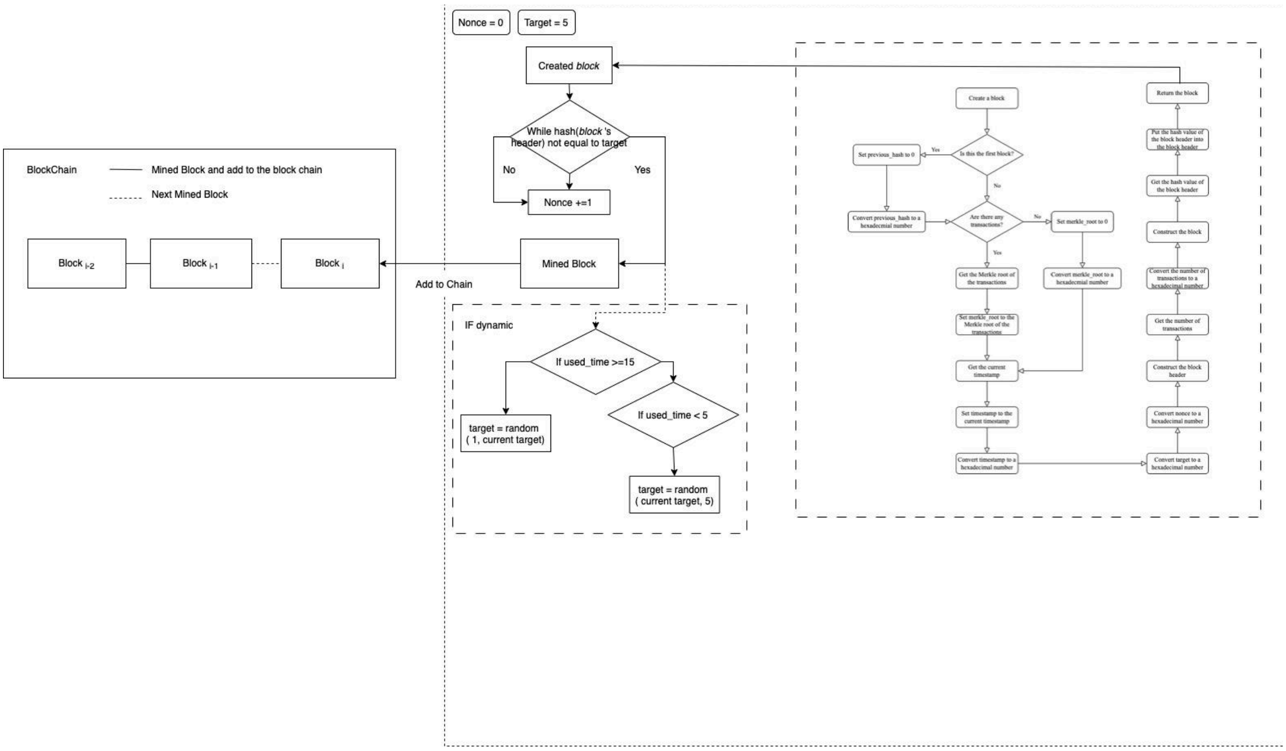
This group project is about generating a mini blockchain using code programming. Building a full blockchain requires complex models, and for this reason, in this project we will only focus on some basic structures. The specific content includes the generation of transaction and blockchain, the Merkle tree, finally the integrity verification. We used python as our code language and build this mini blockchain with some libraries about cryptography. Some other library references include hashlib, json, etc., to help generate blockchains. We divided the project into five parts to allow for division of labour and eventual code integration. The overall code contains several function methods and the final test function, which meets the project requirements and generates the blockchain.

There are five members in the group. The table below shows the details of our group members and the specific division of labour for each person.

2 Related Library and Tools

In 4.1, cryptography module is mainly used. The cryptography module is the core module for cryptographic operations in Python. It provides various cryptographic algorithms, key generation, encryption, decryption, signature and verification.
cryptography.hazmat.primitives. The module contains primitives such as hash functions, symmetric encryption, asymmetric encryption, message authentication codes, etc.***cryptography.hazmat.primitives.asymmetric.*** The module contains asymmetric encryption related primitives, such as RSA, elliptic curve encryption (ECC), etc.
hashlib. The hashlib module provides implementations of common hash algorithms such as MD5, SHA-1, SHA-256, etc.
json. It is mainly used to encode python objects into json format output or storage, and decode json format objects into python objects.
time. The time module provides various time-related functions.
random. The module implements pseudo-random number generators.

3 Pipeline flow of your blockchain system



Nonce = 0
Target = 5
Created *block*
While hash(*block*'s header) not equal to target
No
Yes
Nonce +=1
Mined Block
Add to Chain
BlockChain
Mined Block and add to the block chain
Next Mined Block
Block $_{i-2}$
Block $_{i-1}$
Block $_{i}$
IF dynamic
If used_time >=15
If used_time < 5
target = random (1, current target)
target = random (current target, 5)
Create a block
Is this the first block?
Yes
Set previous_hash to 0
Convert previous_hash to a hexadecmial number
No
Are there any transactions?
No
Set merkle_root to 0
Convert merkle_root to a hexadecmial number
Yes
Get the Merkle root of the transactions
Set merkle_root to the Merkle root of the transactions
Get the current timestamp
Set timestamp to the current timestamp
Convert timestamp to a hexadecimal number
Convert target to a hexadecimal number
Convert nonce to a hexadecimal number
Construct the block header
Get the number of transactions
Convert the number of transactions to a hexadecimal number
Construct the block
Get the hash value of the block header
Put the hash value of the block header into the block header
Return the block

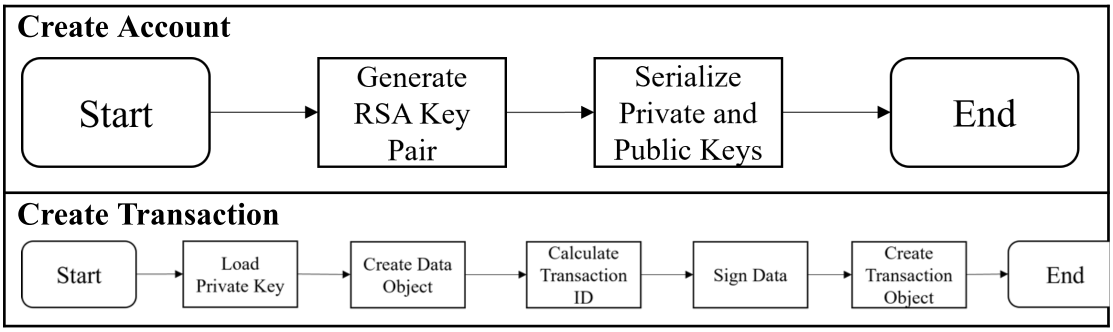
Graph 1: Pipeline of BlockChain System

4 Design and implementation details

4.1 Transaction Generation

For the task 4.1 Transaction Generation, there are two main aspects of the task. First part is about to create a blockchain account for the user. This step is the foundation of the whole project. This section produces the first data structure for task 4.1, which is the public-private key pair. The public key will be used as the account of the user. To implement this, we used the Python cryptography libraries and the RSA algorithm.
Then the next step is to creating the transaction. For this step, we need to consider about the transaction ID, data, the input and the output.

The transaction ID is calculated by taking a hash of the transaction contents. In this case, we imported another Python library, the hashlib, which is used to provide implementations of common hash algorithms. We calculated the SHA256 of this library as the transaction ID. The data is generated as the same way.
For data structures in this part of the project, RSA Key Pair, Transaction Data, Transaction are generated.



Graph 2: Working flow of 4.1

4.2 Verifiable Merkle Tree

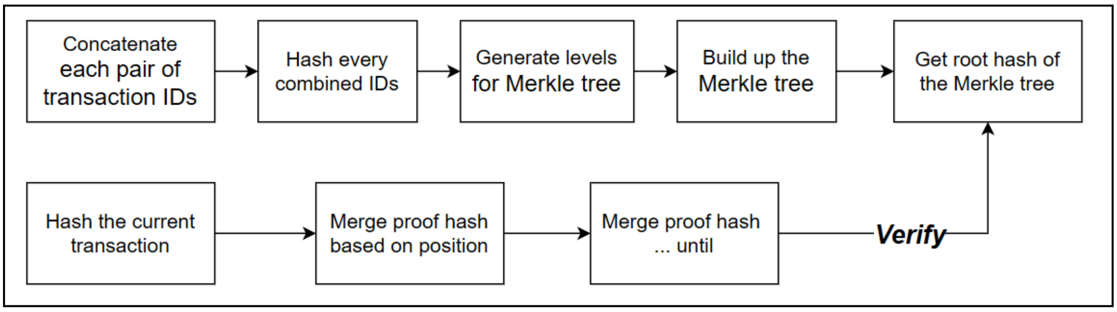
On the basis of creating accounts and transactions in 4.1, we aim to build a Merkle tree for the transactions in 4.2 implementation.

As the core element for every block in a blockchain, the Merkle tree is a data structure used in computer science and cryptography, which is particularly useful for verifying the integrity and consistency of data. It is a binary tree where each leaf node represents a hash of a block of data, and each non-leaf node is a hash of its children. This structure is used to ensure that data blocks received from other peers in a peer-to-peer network are undamaged and unaltered by the root of every tree.

Here we use the SHA256 encryption function in the ***hashlib*** to generate the unique hash value with a fixed length. For each layer, we concatenate every two transactions and then get its hash value, above which we add a new layer. Due to the property of binary tree, we can make a full tree if the number of nodes(transactions) equals the multiple of 2. Otherwise, the left one becomes a separate layer.

After finishing the construction of the Merkle tree for all transactions, we have a set of hash values including the root hash value. When it comes to verifying a single transaction in this block, we just need to combine the hash value of the current node(transaction) and its sibling node until the final hash value equals the root hash value we get previously. Then the integrity of this transaction is proved.

Here is the overview of how this process works.


Concatenate each pair of transaction IDs
Hash every combined IDs
Generate levels for Merkle tree
Build up the Merkle tree
Get root hash of the Merkle tree
Hash the current transaction
Merge proof hash based on position
Merge proof hash ... until
Verify

Graph 3: Working flow of 4.2

4.3. Construction of Blockchain

After creating transactions and building a Merkle tree for them, a block can be created. In our simulation, each block is represented as a Python dictionary. The block header is a nested Python dictionary within the block, and the transactions are stored as a list within the block dictionary. All values in the block header, as well as the number of transactions, are stored as hexadecimal values in big-endian format and are represented as Python strings. Table 1 provides details of each element in the block header, which totals 108 bytes in size. Table 2 provides details of each element in the block, with the block being at least 109 bytes in size.

Elements of a Block Header

Name	Description	Size
previous_hash	The SHA256 hash value of the block header of the previous block	32 bytes
merkle_root	The root of the Merkle tree of the current block	32 bytes
timestamp	The time at which the current block is created, measured as the number of seconds since epoch	4 bytes
target	The number of zeros that *current_hash* must begin with	4 bytes
nonce	A random number for mining a block	4 bytes
current_hash	The SHA256 hash value of the above block header elements	32 bytes

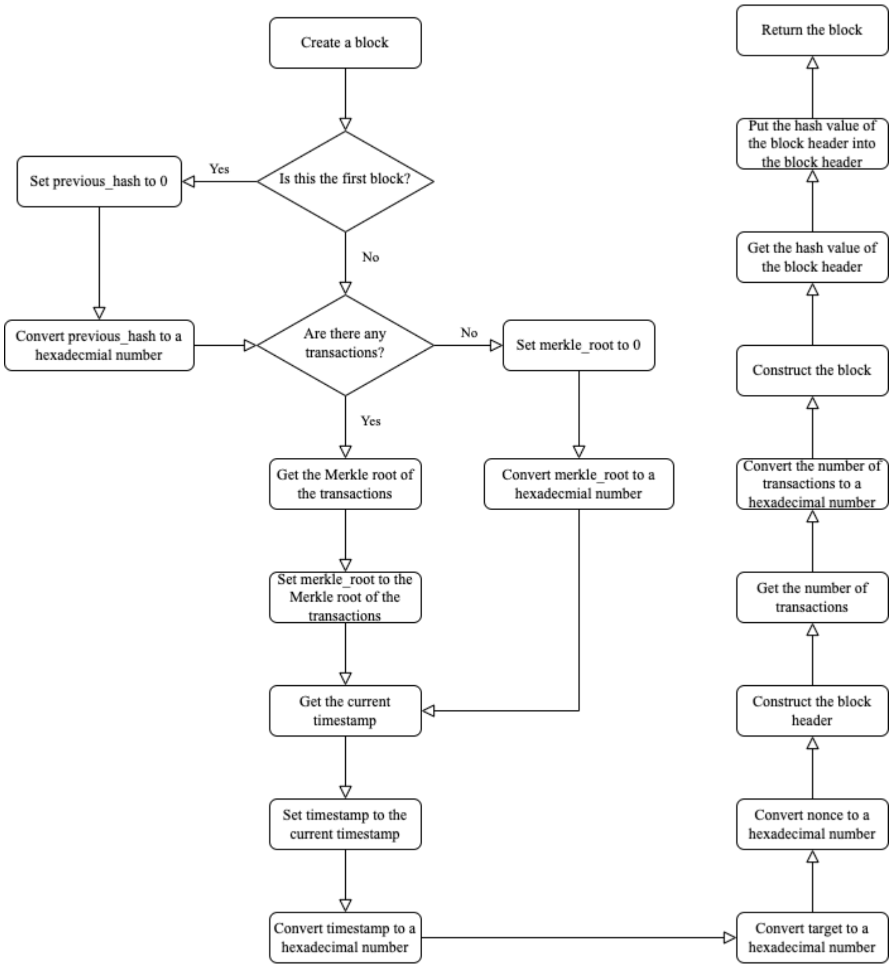
Table 1: Elements of a block header

Elements of a Block

Name	Description	Size
header	The block header, stored as a Python dictionary	108 bytes
number_of_transactions	The number of transactions in the current block	1 byte
transactions	All the transactions in the current block, stored as a Python list	Dependent on *number_of_transactions*

Table 2: Elements of a block

The *create_block* function is responsible for creating a block. The inputs to the function are *previous_hash*, *transactions*, *target*, and *nonce*. The default value for *previous_hash* is *None*. A *None* value for *previous_hash* implies that there is no SHA256 hash value for the previous block. In such cases, the *previous_hash* variable is set to sixty-four zeros. The default value for *transactions* is an empty list, indicating that there are no transactions in the block. If the block contains no transactions, the *merkle_root* variable is set to sixty-four zeros. Otherwise, the *merkle_root* variable is set to the Merkle root of the transactions, which can be obtained from Section 4.2. The *timestamp* variable is set to the current timestamp obtained from the ***time*** library. It represents the time at which the block is created. The *timestamp* is converted to a hexadecimal number, and if its length is less than eight characters, zeros are padded at the front to ensure it spans eight characters. The default values for the *target* and *nonce* variables are zero. These are also converted to hexadecimal numbers, and if their lengths are less than eight characters, zeros are padded at the front to ensure they span eight characters. The block header dictionary is then constructed. The *number_of_transactions* variable is set to the length of the *transactions* list, converted to a hexadecimal number, and padded with zeros at the front if its length is less than two characters to ensure it spans two characters. The block dictionary is then constructed, with the *current_hash* variable set to the SHA256 hash value of the block header, which is stored in the block header. Finally, the block dictionary is returned. Graph 3 shows the workflow of the function for creating a block.


Create a block
Is this the first block?
Yes
Set previous_hash to 0
No
Convert previous_hash to a hexadecmial number
Are there any transactions?
No
Set merkle_root to 0
Yes
Get the Merkle root of the transactions
Convert merkle_root to a hexadecmial number
Set merkle_root to the Merkle root of the transactions
Get the current timestamp
Set timestamp to the current timestamp
Convert timestamp to a hexadecimal number
Convert target to a hexadecimal number
Convert nonce to a hexadecimal number
Construct the block header
Get the number of transactions
Convert the number of transactions to a hexadecimal number
Construct the block
Get the hash value of the block header
Put the hash value of the block header into the block header
Return the block

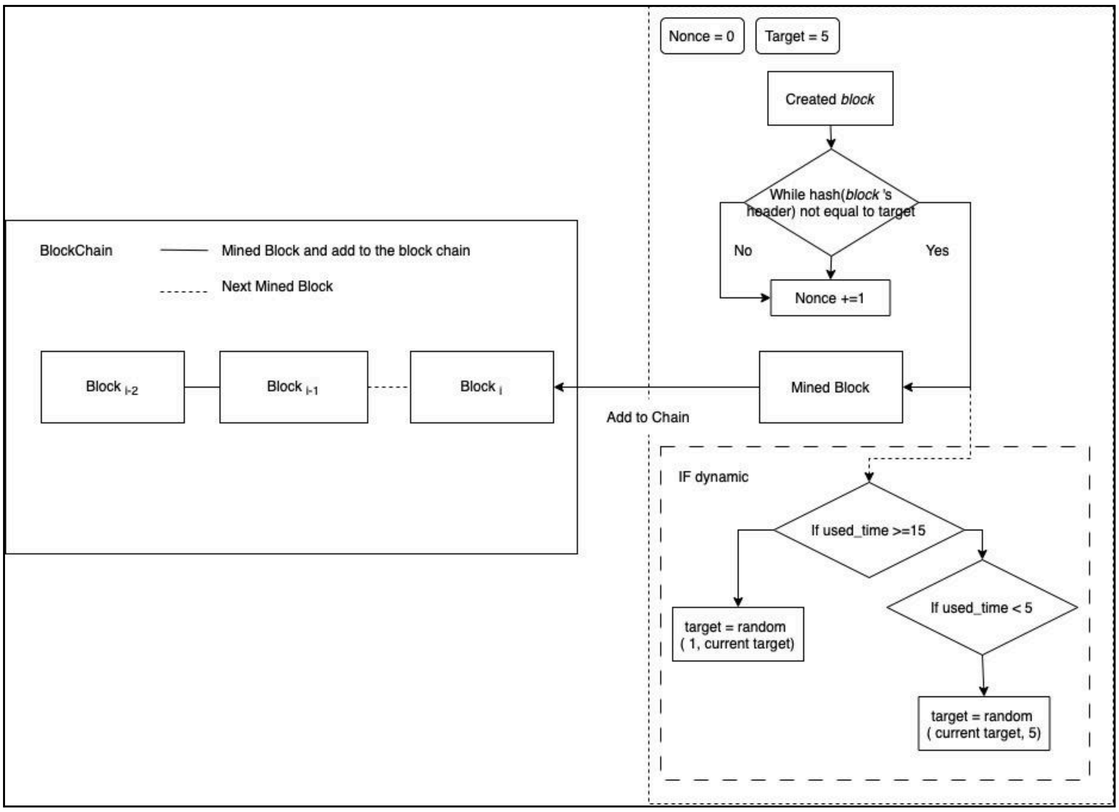
Graph 4: Workflow of 4.3

4.4 Mining a block

In part 4.4, we would like to simulate the process of mining a block, which is implemented by creating a class called BlockChain. Basically, the class maintains a list (also called a chain) that stores the blocks generated in 4.3. The process of mining involves finding a nonce which meetes a specified difficulty target. By defaut, the target would be the header start at 5 “0”. To simulate the real environment of blockchain, I also wrote a function called *change_target,* which could arrnge target if the time used in the mining part.

At the beginning, a nonce would be initialized and time would be recorded after a block is created. The nonce would increase until the hash of the header meets the target. Once a valid block is found, time would be recorded. Both the running time and the block would return. This process illustrates how the mining process secures the network by ensuring the blocks are minded correctly before appended to the blockchian.

Additionally, the difficulty of mining a block would be adjusted if the Blockchain is dynamic. The change target method modifies the target based on the time used in mining the last block. If the mining takes too long, which is 15 seconds, the function will make the target easier to mine future blocks. In contrast, if mining is too quick, which takes less than 5 seconds, the difficulty is reduced. This code provide a fundamental understanding of how blockchain and mining process run in a simpler manner.



Graph 5: Workflow of 4.4

4.5 Integrity Verification

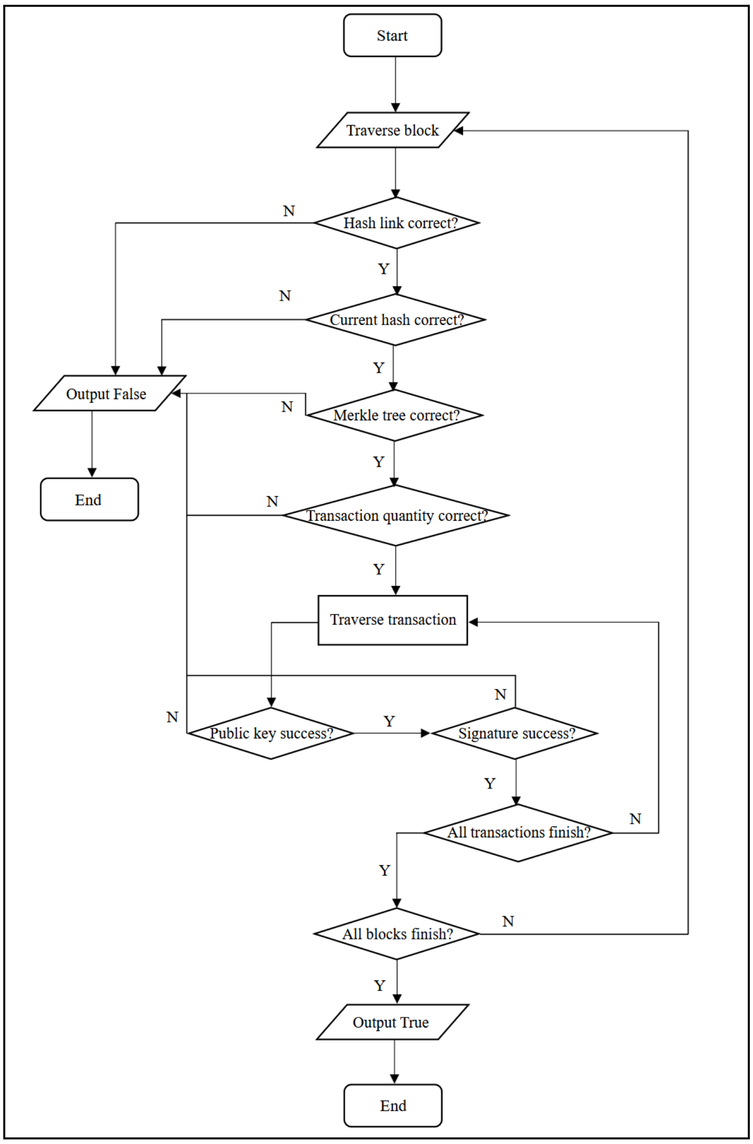
The blockchain integrity verification process starts from receiving input from the blockchain instance. The system first initializes the verification environment, loads the necessary encryption components (based on the key generation module in 4.1 and the Merkle tree construction function in 4.2), and then traverses each block in the chain in order (starting from the first non genesis block) to perform hierarchical progressive verification. The verification process includes the following steps in sequence [2, 3, 4]:

1. Hash chain continuity verification: Check whether the ‘preview_ hash’ of the current block strictly matches the ‘current hash’ of the previous block. If a fracture is detected (such as tampering with a historical block), the process will be terminated immediately and error location information will be returned;
2. Current hash value reconstruction: According to the block construction rules in 4.3, recalculate the hash value from the original header data (excluding the ‘current hash’ field), compare it with the stored value of the block header, and ensure that the block header data has not been tampered with;
3. Merkle tree root consistency verification: Call the Merkle tree generation algorithm in 4.2 to reconstruct the root hash based on the transaction list, compare it with the ‘merkle_root’ declared in the block header, and check the integrity of the transaction data;
4. Transaction quantity verification: Verify the consistency between the actual transaction quantity and the field ‘numb_of_transactions’ in the block header to

prevent transaction concealment or forgery;

5. Transaction by transaction signature verification: For each transaction, parse the sender's public key (PEM format key in 4.1), reconstruct the data according to the original signature rules (JSON key sorting, UTF-8 encoding), and use RSA-PSS algorithm to verify the validity of the signature. Any transaction verification failure triggers the termination of the process and records the specific transaction ID.

The exception handling mechanism adopts a hierarchical response strategy: hash mismatch class errors (such as block hash chain breakage, Merkle root inconsistency) are directly returned to the block level error location; Fine grained anomalies such as invalid transaction signatures are recorded along with the transaction ID and original data, generating structured logs (including timestamps, anomaly types, and location information) that support precise auditing and traceability. The entire process is designed as an atomized operation, and if any verification step fails, the subsequent process will be terminated immediately to ensure efficient resource utilization and security.



Graph 6: Workflow of 4.5

5 Experimental results

1. Output of test code 4.5.1 [6]

```
=== Test 1 - Original chain verification ===
The validity of blockchain: True

=== Test 2 - Current Hash tampering test ===
Block 1 has invalid current hash
Validity after tampering: False

=== Test 3 - Merkle tree root tampering test ===
Block 1 has invalid current hash
Validity after tampering: False

=== Test 4 - Transaction Signature Tampering Test ===
Transaction
01fe24f970bb35c7284dd16158ec140a5da820d0cbf67b2173e94dc9a7715250
verification failed:
Validity after tampering: False

=== Test 5 - Recovery verification ===
Validity after recovery: True
```

6 Conclusion

Our project successfully implemented a mini blockchain system demonstrating core principles through four key components: secure transaction generation with digital signatures, an efficient Merkle tree for transaction verification, immutable block construction, and comprehensive integrity validation. Our Python implementation leveraged cryptographic hashing (SHA-256) and public-key infrastructure (RSA) to ensure data authenticity and tamper-resistance, with all components rigorously tested for correctness. The system's modular design enabled effective team collaboration while maintaining technical coherence.

While fulfilling all project requirements, our implementation lays the groundwork for potential enhancements like consensus algorithms and network decentralization. The accompanying video presentation further demonstrates the system's functionality and our development process, confirming the practical viability of the designed solution. Our project has not only validated fundamental blockchain concepts but also highlighted the importance of cryptographic techniques in distributed systems.

References

[1] Yaga, D., Mell, P., Roby, N., & Scarfone, K. (2018). Blockchain Technology Overview. *National Institute of Standards and Technology*, *1*(1). https://doi.org/10.6028/nist.ir.8202

[2] Python Language Reference. (2023). 'hashlib — Secure hashes and message digests'. Retrieved from https://docs.python.org/3/library/hashlib.html

[3] Python Language Reference. (2023). 'json — JSON encoder and decoder'. Retrieved from https://docs.python.org/3/library/json.html

[4] Cryptography. (2023). 'Cryptography: Cryptographic recipes and primitives for Python'. Retrieved from https://cryptography.io/

[5] Python Language Reference. (2023). 'time — Time access and conversions'. Retrieved

from https://docs.python.org/3/library/time.html

[6] Python Language Reference. (2023). ‘random — Generate pseudo-random numbers’. Retrieved from https://docs.python.org/3/library/random.html

Appendics

4.1 Transaction Generation

```
Function create_account():
   Generate RSA private_key and public_key
   Serialize private_key to PEM format
   Serialize public_key to PEM format
   Return private_pem, public_pem

Function create_transaction():
   Load sender_private_key from PEM
   Create data object with amount, sender, and receiver
   Calculate transaction_id as SHA256 hash of data
   Sign data with sender_private_key using PSS and SHA256
   Create transaction object with transaction_id, data, and signature
   Return transaction
```

4.2 Verifiable Merkle Tree

```
Function sha256_hash(data):
     Return SHA256 hash of data (UTF-8 encoded) as hex string
Function build_merkle_tree(transactions):
     If single transaction remains:
          Return the transaction
     Initialize new_level as empty list
     For each pair of transactions:
          Concatenate their transaction_ids
          Append hash of concatenation to new_level
     If odd transaction count:
          Append last transaction to new_level
     Return final node
Function get_merkle_root(transactions):
     If no transactions:
          Return empty string
     Return transaction_id from build_merkle_tree result
Function verify_merkle_proof(transaction, proof, merkle_root):
     Set current_hash to transaction's transaction_id
     For each proof step:
          If proof position is left:
               Update current_hash with proof_hash + current_hash
          Else:
               Update current_hash with current_hash + proof_hash
     Return True if final hash matches merkle_root
```

4.3. Construction of Blockchain

```
Function create_block(previous_hash, transactions, target, nonce):
   If this block is the first block:
      Set previous_hash as 0
      Convert previous_hash to a hexadecimal number
```

```
    If there are no transactions in this block:
        Set merkle_root as 0
        Convert merkle_root to a hexadecimal number
    Else:
        Get the Merkle root of the transactions
        Set merkle_root as the Merkle root of the transactions
    Get the current timestamp
    Set timestamp as the current timestamp
    Convert the timestamp to a hexadecimal number
    Convert the target to a hexadecimal number
    Convert the nonce to a hexadecimal number
    Construct the block header by combining the previous_hash, the merkle_root, the timestamp, the target, and the nonce to form a dictionary object
    Get the number of transactions
    Convert the number of transactions to a hexadecimal number
    Construct the block by combining the header, the number of transactions, and the list of transactions to form a dictionary object
    Get the hash value of the block header
    Put the hash value of the block header into the block header
    Return the block
```

4.4 Mining a block

```
CLASS BlockChain:

    Function __init__():
        Init chain AS empty list
        Init target AS 5 # default target is 5

    Function mine_block(transactions):
        Init nonce AS 0 # Default nonce is 0
        Init start_time AS current time
        Init block AS create_block(previous_hash, transactions, target, nonce)

        WHILE block.header.current_hash[0:target] NOT EQUAL TO '0' * target:
            Increment nonce by 1
            block = create_block(previous_hash, transactions, target, nonce)

        used_time = current time - start_time
        RETURN block, used_time

    Function change_target(time):
        If time >= 15:
            Set target AS random integer between 1 and current target
        Else IF time < 5:
            Set target AS random integer between current target and 5

    Function  add_block2chain(transactions, dynamic):
        mined_block, time = mine_block(transactions)
        Append mined_block to chain

        If dynamic is TRUE:
            Call change_target(time)
```

4.5 Integrity Verification

```
function is_chain_valid(chain):
    for i from 1 to len(chain)-1:
        current_block = chain[i]
        prev_block = chain[i-1]

        # Verify hash link
        if current_block.header.previous_hash != prev_block.header.current_hash:
            return False

        # Verify the current hash
        raw_header = {
            'previous_hash': current_block.header.previous_hash,
            'merkle_root': current_block.header.merkle_root,
            'timestamp': current_block.header.timestamp,
            'target': current_block.header.target,
            'nonce': current_block.header.nonce
        }
        if sha256(str(raw_header)) != current_block.header.current_hash:
            return False

        # Verify Merkle tree root
 if compute_merkle_root(current_block.transactions) !=
current_block.header.merkle_root:
            return False

        # Verify the number of transactions
        if len(current_block.transactions) !=
int(current_block.number_of_transactions):
            return False

        # Verify transaction signature
        for tx in current_block.transactions:
            if not verify_signature(tx.data, tx.signature, tx.sender_public_key):
                return False
    return True
```

Appendix 1 : Pseudo code