

# Blockchain

## Assignment 2

**Sir** Abid Rauf



Ali Hamza 20I-1881  
Abdullah Malik 20I-0930

8 October, 2023

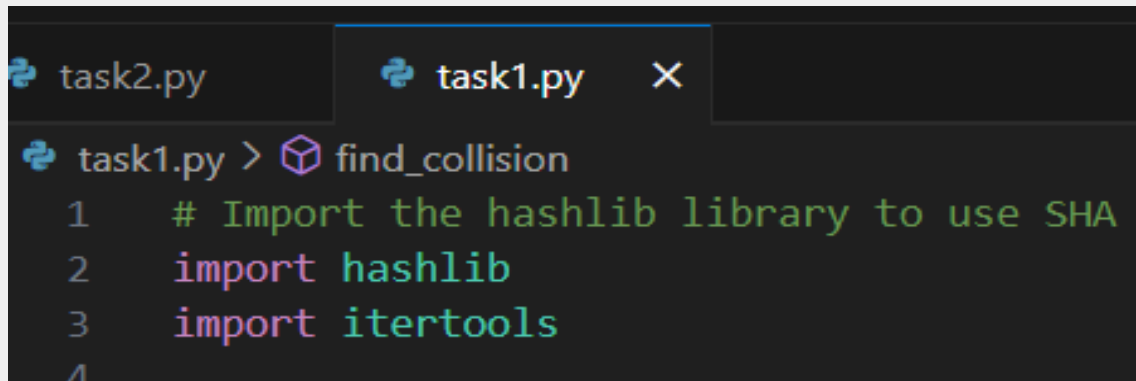
# Problems

## Problem:

- 1- For this coding assignment, you'll be producing a hash collision in a new hash function we'll call SHA0.125. This hash function is just defined as SHA256 truncated to the first 4 bytes (the first 8 hex digits). Thus, SHA0.125 only has a digest size of 32 bits (1/8 of SHA256). You should be able to produce a collision in no more than a couple seconds.
- 2- Merkle trees are hash-based data structures used to prove the integrity of transaction data stored in the block. For this exercise you may assume that all trees are binary, balanced, and that the number of transactions to be stored are some exponent of two. Your transactions can be of any size. You'll be writing your own implementation of a Merkle tree. You'll then be writing code to verify Merkle proofs.  
Save your code from this, as you may find it useful for your cryptocurrency project.

## Task 1

### Importing Libraries:



```
task2.py task1.py X
task1.py > find_collision
1  # Import the hashlib library to use SHA
2  import hashlib
3  import itertools
4
```

#### Here, we import two libraries:

**hashlib:** This library provides various hash functions, including SHA-256, which we will use.

**itertools:** This library allows us to generate combinations of characters for input data.

## Defining the SHA0.125 Hash Function:

```
# Define a function to calculate a truncated SHA0.125 hash from input data.
def sha0_125(input_data):
    # Calculate the SHA256 hash of the input data.
    sha256_hash = hashlib.sha256(input_data.encode()).hexdigest()

    # Truncate it to the first 4 bytes (8 hex digits).
    truncated_hash = sha256_hash[:8]

    return truncated_hash
```

This function, **sha0\_125**, takes an **input\_data** string, calculates its SHA-256 hash, and then truncates it to the first 4 bytes (8 hex digits). This truncation is what makes it a "SHA0.125" hash.

## Finding Hash Collisions:

```
# Define a function to find a collision in the truncated hashes.
def find_collision():
    # Create a dictionary to store hash results mapped to input data.
    hash_to_input = {}

    # Get the character set and input length from the user.
    input_chars = input("Enter the character set: ") # Characters to use in input data.
    input_length = int(input("Enter the input length: ")) # Length of input data.

    # Generate all possible combinations of input strings using the provided character set and length.
    for combination in itertools.product(input_chars, repeat=input_length):
        input_data = "".join(combination)

        # Calculate the SHA0.125 hash for the current input data.
        hash_result = sha0_125(input_data)

        # Check if this hash has been seen before and if it's not the same as the current input.
        if hash_result in hash_to_input and hash_to_input[hash_result] != input_data:
            # A collision is found! Print the details.
            print("Collision found!")
            print("Input 1:", hash_to_input[hash_result])
            print("Hash 1:", hash_result)
            print("Input 2:", input_data)
            print("Hash 2:", sha0_125(input_data))
            return
        else:
            # Store the hash result mapped to the current input data.
            hash_to_input[hash_result] = input_data
```

This function, **find\_collision**, does the following:

- It initializes an empty dictionary, **hash\_to\_input**, to store hash results mapped to input data.
- It prompts the user to enter a character set and the desired input length.



```

# Import the hashlib library to use SHA hashing and itertools for
combinations.
import hashlib
import itertools

# Define a function to calculate a truncated SHA0.125 hash from input
data.
def sha0_125(input_data):
    # Calculate the SHA256 hash of the input data.
    sha256_hash = hashlib.sha256(input_data.encode()).hexdigest()

    # Truncate it to the first 4 bytes (8 hex digits).
    truncated_hash = sha256_hash[:8]

    return truncated_hash

# Define a function to find a collision in the truncated hashes.
def find_collision():
    # Create a dictionary to store hash results mapped to input data.
    hash_to_input = {}

    # Get the character set and input length from the user.
    input_chars = input("Enter the character set: ") # Characters to
use in input data.
    input_length = int(input("Enter the input length: ")) # Length of
input data.

    # Generate all possible combinations of input strings using the
provided character set and length.
    for combination in itertools.product(input_chars,
repeat=input_length):
        input_data = "".join(combination)

        # Calculate the SHA0.125 hash for the current input data.
        hash_result = sha0_125(input_data)

        # Check if this hash has been seen before and if it's not the
same as the current input.
        if hash_result in hash_to_input and hash_to_input[hash_result]
!= input_data:
            # A collision is found! Print the details.
            print("Collision found!")
            print("Input 1:", hash_to_input[hash_result])
            print("Hash 1:", hash_result)
            print("Input 2:", input_data)
            print("Hash 2:", sha0_125(input_data))
            return
        else:
            # Store the hash result mapped to the current input data.
            hash_to_input[hash_result] = input_data

if __name__ == "__main__":
    # Call the find_collision function to search for hash collisions.
    find_collision()

```

## Task 2

### Defining the Hash Function:

```
3
4 # Define a function to calculate the SHA-256 hash of a given value.
5 def hash_function(value):
6     return hashlib.sha256(value.encode()).hexdigest()
7
```

Here, a function named **hash\_function** is defined. It takes a **value** as input, encodes it, calculates the SHA-256 hash, and returns the hexadecimal representation of the hash.

### Building a Merkle Tree:

```
7
8 # Define a function to build a Merkle tree from a list of transactions.
9 def build_merkle_tree(transactions):
10     # Calculate the hash of each transaction and store them in a list.
11     tree = [hash_function(transaction) for transaction in transactions]
12
13     # Continue hashing pairs of transactions until there's only one root hash left.
14     while len(tree) > 1:
15         tree = [hash_function(tree[i] + tree[i + 1]) for i in range(0, len(tree), 2)]
16
17     # Return the root hash of the Merkle tree.
18     return tree[0]
19
```

This function, **build\_merkle\_tree**, takes a list of **transactions** as input. It starts by calculating the hash of each transaction and stores these hashes in a list called **tree**. It then repeatedly combines pairs of hashes until only one root hash remains, which is returned as the result.

## Printing the Merkle Tree Structure:

```

19
20 # Define a function to print the structure of the Merkle tree.
21 def print_tree_structure(tree, indent=""):
22     # If there's only one hash, print it with a "+" symbol.
23     if len(tree) == 1:
24         print(indent + "+-- " + tree[0])
25     else:
26         # If there are more hashes, split them into left and right subtrees and print them.
27         mid = len(tree) // 2
28         left_tree = tree[:mid]
29         right_tree = tree[mid:]
30
31         # Print the current hash with a "|" symbol and recurse into left and right subtrees.
32         print(indent + "|-- " + tree[0])
33         print_tree_structure(left_tree, indent + "| ")
34         print_tree_structure(right_tree, indent + "| ")
35

```

This function, **print\_tree\_structure**, is used to print the structure of the Merkle tree. It takes the **tree** (list of hashes) and an optional **indent** parameter to control the formatting. It prints each level of the tree with appropriate indentation.

## Generating Dummy Transactions:

```

35
36 # Define a function to generate dummy transactions based on the given number.
37 def generate_dummy_transactions(num_transactions):
38     return [f"Tx{i}" for i in range(1, num_transactions+1)]
39

```

**generate\_dummy\_transactions** generates a list of dummy transaction names based on the provided **num\_transactions**.

## Generating a Proof:

```

39
40 # Define a function to generate a proof for a target transaction in the Merkle tree.
41 def generate_proof(transactions, target_transaction):
42     # Check if the target transaction exists in the list of transactions.
43     if target_transaction not in transactions:
44         return ["Transaction not found"]
45
46     # Find the index of the target transaction.
47     target_index = transactions.index(target_transaction)
48
49     # Calculate the hash of all transactions and build the proof list.
50     tree = [hash_function(transaction) for transaction in transactions]
51     proof = []
52
53     while len(tree) > 1:
54         # Determine the sibling hash and add it to the proof.
55         if target_index % 2 == 0:
56             sibling = tree[target_index + 1]
57         else:
58             sibling = tree[target_index - 1]
59
60         proof.append(sibling)
61         target_index = target_index // 2
62
63         # Recalculate the tree by hashing pairs of hashes.
64         tree = [hash_function(tree[i] + tree[i + 1]) for i in range(0, len(tree), 2)]
65
66     # Return the proof for the target transaction.
67     return proof
68

```

This function, **generate\_proof**, takes a list of **transactions** and a **target\_transaction** as input. It checks if the target transaction exists in the list and, if found, generates a Merkle proof for that transaction.

## Verifying a Proof:

```

68
69 # Define a function to verify if a given proof is valid.
70 def verify_proof(root_hash, target_transaction, proof):
71     computed_hash = hash_function(target_transaction)
72
73     for proof_hash in proof:
74         # Determine the order of concatenation and recompute the hash.
75         if computed_hash < proof_hash:
76             combined = computed_hash + proof_hash
77         else:
78             combined = proof_hash + computed_hash
79         computed_hash = hash_function(combined)
80
81     # Check if the computed hash matches the root hash.
82     return computed_hash == root_hash
83

```



**verify\_proof** is used to verify if a given Merkle proof is valid by comparing the computed hash with the root hash.

## User Input and Execution:

```
83
84 # Take the number of transactions as input from the user.
85 num_transactions = int(input("Enter the number of transactions: "))
86
87 # Generate dummy transactions based on the user's input.
88 transactions = generate_dummy_transactions(num_transactions)
89
90 # Build the Merkle tree and calculate its root hash.
91 root_hash = build_merkle_tree(transactions)
92
93 # Take the target transaction as input from the user.
94 target_transaction = input("Enter the target transaction: ")
95
96 # Generate a proof for the target transaction.
97 proof = generate_proof(transactions, target_transaction)
98
99 # Print the structure of the Merkle tree.
100 print("\nMerkle Tree Structure:")
101 print_tree_structure([hash_function(transaction) for transaction in transactions])
102
103 # Verify the proof and check if it's valid.
104 is_valid = verify_proof(root_hash, target_transaction, proof)
105
106 # Display the results to the user.
107 print(f"\nRoot hash: {root_hash}")
108 print(f"Target transaction: {target_transaction}")
109 print(f"Target transaction Hash: {hash_function(target_transaction)}")
110
111 print("\nMerkle proof:")
112 for i, item in enumerate(proof):
113     print(f"Level {i+1}: {item}")
114
115 print(f"Is valid proof: {is_valid}")
116
```

The code starts by taking user input for the number of transactions and then generates dummy transactions based on this input. It builds the Merkle tree, calculates the root hash, and takes the target transaction as input from the user. After generating a proof and verifying it, the code prints the Merkle tree structure and the results.

## Output Tree:

```
PS E:\Semester#7\Assignments\Blockchain\Assignment 2> & "C:/Users/Ali Hamza/AppData/Local/Programs/Python/Python311/python.exe
ssignment 2/task2.py"
Enter the number of transactions: 8
Enter the target transaction: Tx6

Merkle Tree Structure:
|-- 55f743d0d1b9bd86bbd96a46ba4272ddde19f09e3f6e47832e34bb2779a120b5
|   |-- 55f743d0d1b9bd86bbd96a46ba4272ddde19f09e3f6e47832e34bb2779a120b5
|   |   |-- 55f743d0d1b9bd86bbd96a46ba4272ddde19f09e3f6e47832e34bb2779a120b5
|   |   |-- 80ed43f7a11b3295850dd90cc0cfc9a80334f433af8d3d88a1c5e78aff14988f
|   |   |-- 13288c2ba4bbc9af05aa9ccd39b0cc603dc9e30471d97565c9ef3c3604b7ca23
|   |   |-- 13288c2ba4bbc9af05aa9ccd39b0cc603dc9e30471d97565c9ef3c3604b7ca23
|   |   |-- 75af2038a4bcf0230372d08b917047bdcbad80e5f130061bd5b31596df174b67
|   |-- 68ef264fc4289f52448d9d44991745c9be7ef3f0216bf66a580f4edb69543bdd
|   |   |-- 68ef264fc4289f52448d9d44991745c9be7ef3f0216bf66a580f4edb69543bdd
|   |   |-- 94431974160bbcc8c2fc176d9cc3c4291a0892198e39fb36b208786d2145e895
|   |   |-- 54f33bb3bff6b1062846211f1b35ea72730889b17f741e772328ff3462459bbd
|   |   |-- 54f33bb3bff6b1062846211f1b35ea72730889b17f741e772328ff3462459bbd
|   |   |-- 602b6aea5bf928cd9b98e111b17012ef2009bb49b3bcd6421951edd7a348205f

Root hash: 2f115d7f1e298eb7bd1cba2e29994a368dcb5f40f108460056c05d315276eb9d
Target transaction: Tx6
Target transaction Hash: 94431974160bbcc8c2fc176d9cc3c4291a0892198e39fb36b208786d2145e895
```

## Proof:

```
Root hash: 2f115d7f1e298eb7bd1cba2e29994a368dcb5f40f108460056c05d315276eb9d
Target transaction: Tx6
Target transaction Hash: 94431974160bbcc8c2fc176d9cc3c4291a0892198e39fb36b208786d2145e895

Merkle proof:
Level 1: 68ef264fc4289f52448d9d44991745c9be7ef3f0216bf66a580f4edb69543bdd
Level 2: 4ec8ed73872d5a5b9afc9c9777df161d180784791b881d9a7bd37b88f6a7552
Level 3: 5b260dbcbff182d10cdbd21d8cb9e4446fe71820bb91c8dced8dcfd0e8a9c8ac
Is valid proof: True
```

## CODE

```

# Import the hashlib library to use SHA-256 hashing.
import hashlib

# Define a function to calculate the SHA-256 hash of a given value.
def hash_function(value):
    return hashlib.sha256(value.encode()).hexdigest()

# Define a function to build a Merkle tree from a list of transactions.
def build_merkle_tree(transactions):
    # Calculate the hash of each transaction and store them in a list.
    tree = [hash_function(transaction) for transaction in transactions]

    # Continue hashing pairs of transactions until there's only one root
    hash left.
    while len(tree) > 1:
        tree = [hash_function(tree[i] + tree[i + 1]) for i in range(0,
len(tree), 2)]

    # Return the root hash of the Merkle tree.
    return tree[0]

# Define a function to print the structure of the Merkle tree.
def print_tree_structure(tree, indent=""):
    # If there's only one hash, print it with a "+" symbol.
    if len(tree) == 1:
        print(indent + "+-- " + tree[0])
    else:
        # If there are more hashes, split them into left and right
        subtrees and print them.
        mid = len(tree) // 2
        left_tree = tree[:mid]
        right_tree = tree[mid:]

        # Print the current hash with a "/" symbol and recurse into left
        and right subtrees.
        print(indent + "|-- " + tree[0])
        print_tree_structure(left_tree, indent + "| ")
        print_tree_structure(right_tree, indent + "| ")

```

```

# Define a function to generate dummy transactions based on the given
number.
def generate_dummy_transactions(num_transactions):
    return [f"Tx{i}" for i in range(1, num_transactions+1)]

# Define a function to generate a proof for a target transaction in the
Merkle tree.
def generate_proof(transactions, target_transaction):
    # Check if the target transaction exists in the list of
    transactions.
    if target_transaction not in transactions:
        return ["Transaction not found"]

    # Find the index of the target transaction.
    target_index = transactions.index(target_transaction)

    # Calculate the hash of all transactions and build the proof list.
    tree = [hash_function(transaction) for transaction in transactions]
    proof = []

    while len(tree) > 1:
        # Determine the sibling hash and add it to the proof.
        if target_index % 2 == 0:
            sibling = tree[target_index + 1]
        else:
            sibling = tree[target_index - 1]

        proof.append(sibling)
        target_index = target_index // 2

        # Recalculate the tree by hashing pairs of hashes.
        tree = [hash_function(tree[i] + tree[i + 1]) for i in range(0,
len(tree), 2)]

    # Return the proof for the target transaction.
    return proof

```

```

# Define a function to verify if a given proof is valid.
def verify_proof(root_hash, target_transaction, proof):
    computed_hash = hash_function(target_transaction)

    for proof_hash in proof:
        # Determine the order of concatenation and recompute the hash.
        if computed_hash < proof_hash:
            combined = computed_hash + proof_hash
        else:
            combined = proof_hash + computed_hash
        computed_hash = hash_function(combined)

    # Check if the computed hash matches the root hash.
    return computed_hash == root_hash

# Take the number of transactions as input from the user.
num_transactions = int(input("Enter the number of transactions: "))

# Generate dummy transactions based on the user's input.
transactions = generate_dummy_transactions(num_transactions)

# Build the Merkle tree and calculate its root hash.
root_hash = build_merkle_tree(transactions)

# Take the target transaction as input from the user.
target_transaction = input("Enter the target transaction: ")

# Generate a proof for the target transaction.
proof = generate_proof(transactions, target_transaction)

# Print the structure of the Merkle tree.
print("\nMerkle Tree Structure:")
print_tree_structure([hash_function(transaction) for transaction in
transactions])

# Verify the proof and check if it's valid.
is_valid = verify_proof(root_hash, target_transaction, proof)

```

```
# Print the structure of the Merkle tree.
print("\nMerkle Tree Structure:")
print_tree_structure([hash_function(transaction) for transaction in
transactions])

# Verify the proof and check if it's valid.
is_valid = verify_proof(root_hash, target_transaction, proof)

# Display the results to the user.
print(f"\nRoot hash: {root_hash}")
print(f"Target transaction: {target_transaction}")
print(f"Target transaction Hash: {hash_function(target_transaction)}")

print("\nMerkle proof:")
for i, item in enumerate(proof):
    print(f"Level {i+1}: {item}")

print(f"Is valid proof: {is_valid}")
```