Experiment 2

Aim: Write a Python program to understand SHA and Cryptography in Blockchain

Theory: This experiment explores the fundamental cryptographic primitives that underpin blockchain technology: **Secure Hash Algorithms (specifically SHA-256)** and **Merkle Trees**.

1. Cryptographic Hash Functions (SHA-256) A cryptographic hash function is a mathematical algorithm that maps data of arbitrary size (the "message") to a bit string of a fixed size (the "hash" or "digest").

- **SHA-256 (Secure Hash Algorithm 2**
- **56-bit):** This is a member of the SHA-2 family of algorithms, designed by the NSA. It produces a 256-bit (32-byte) hash value, typically rendered as a 64-character hexadecimal string.
- **Key Properties:**
    - **Deterministic:** The same input will always produce the same output.
    - **Pre-image Resistance (One-way):** It is computationally infeasible to determine the original input string given only the hash output.
    - **Avalanche Effect:** A tiny change in the input (e.g., changing "A" to "a") produces a completely different hash output. This is vital for ensuring data integrity in a blockchain.

2. The Nonce and Proof-of-Work (PoW) In blockchain networks like Bitcoin, "mining" is the process of validating transactions and adding them to the chain. This relies on a consensus mechanism called Proof-of-Work.

- **Nonce (Number Used Once):** An arbitrary number that miners change repeatedly.
- **The Puzzle:** Miners combine the block data with a nonce and hash the result. The goal is to find a nonce that results in a hash starting with a specific number of leading zeros (the "difficulty target").
- **Computational Effort:** Because SHA-256 is unpredictable, there is no formula to calculate the correct nonce. Miners must use brute force (guessing millions of nonces per second). This expenditure of computational energy secures the network against spam and attacks.

3. Merkle Trees (Hash Trees) A Merkle Tree is a hierarchical data structure used to verify the integrity of data efficiently.

- **Structure:** It is a binary tree where **leaf nodes** are hashes of individual data blocks (like transactions). **Non-leaf nodes** are hashes of the concatenation of their children.

- **Merkle Root:** The single hash at the top of the tree. This root is stored in the block header.
- **Role in Blockchain:** If any single transaction at the bottom is changed, the hash change propagates up the tree, altering the Merkle Root. This allows a blockchain to easily detect if any transaction in a block has been tampered with.

Tasks Performed:

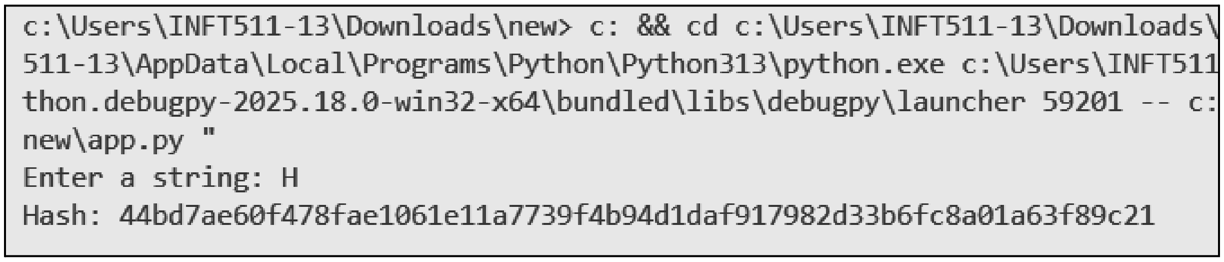
1. **Hash Generation using SHA-256**: Developed a Python program to compute a SHA-256 hash for any given input string using the hashlib library.
2. **Target Hash Generation with Nonce**: Created a program to generate a hash code by concatenating a user input string and a nonce value to simulate the mining process.
3. **Proof-of-Work Puzzle Solving**: Implemented a program to find the nonce that, when combined with a given input string, produces a hash starting with a specified number of leading zeros.
4. **Merkle Tree Construction:** Built a Merkle Tree from a list of transactions by recursively hashing pairs of transaction hashes, doubling up last nodes if needed, and generated the Merkle Root hash for blockchain transaction integrity.

Programs And Outputs:

Program #1: Hash Generation Using hashlib Library

```
import hashlib
 def create_hash(string):
         # Create a hash object using SHA-256 algorithm
         hash_object = hashlib.sha256()
         # Convert the string to bytes and update the hash object
    hash_object.update(string.encode('utf-8'))
         # Get the hexadecimal representation of the hash
         hash_string = hash_object.hexdigest()
         # Return the hash string
         return hash_string
 # Example usage
 input_string = input("Enter a string: ")
 hash_result = create_hash(input_string)
 print("Hash:", hash_result)
```

Output:

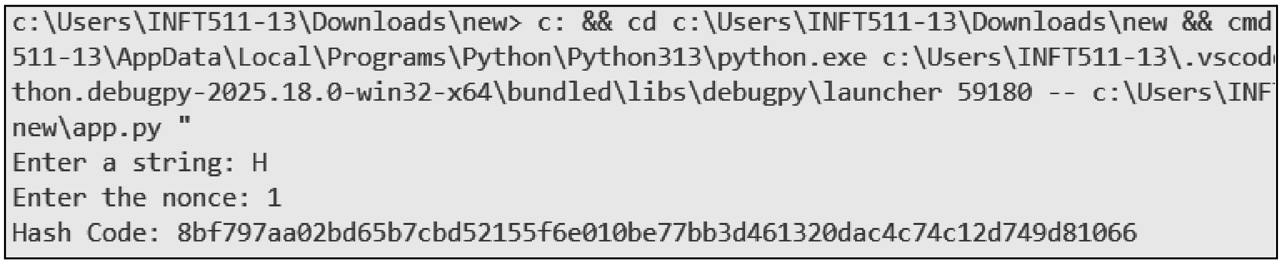


```
c:\Users\INFT511-13\Downloads\new> c: && cd c:\Users\INFT511-13\Downloads\
511-13\AppData\Local\Programs\Python\Python313\python.exe c:\Users\INFT511
thon.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher 59201 -- c:
new\app.py "
Enter a string: H
Hash: 44bd7ae60f478fae1061e11a7739f4b94d1daf917982d33b6fc8a01a63f89c21
```

Program #2: Generating Target Hash with Input String and Nonce

```
import hashlib
 # Get user input
 input_string = input("Enter a string: ")
 nonce = input("Enter the nonce: ")
 # Concatenate the string and nonce
 hash_string = input_string + nonce
 # Calculate the hash using SHA-256
 hash_object = hashlib.sha256(hash_string.encode('utf-8'))
 hash_code = hash_object.hexdigest()
 # Print the hash code
 print("Hash Code:", hash_code)
```

Output:

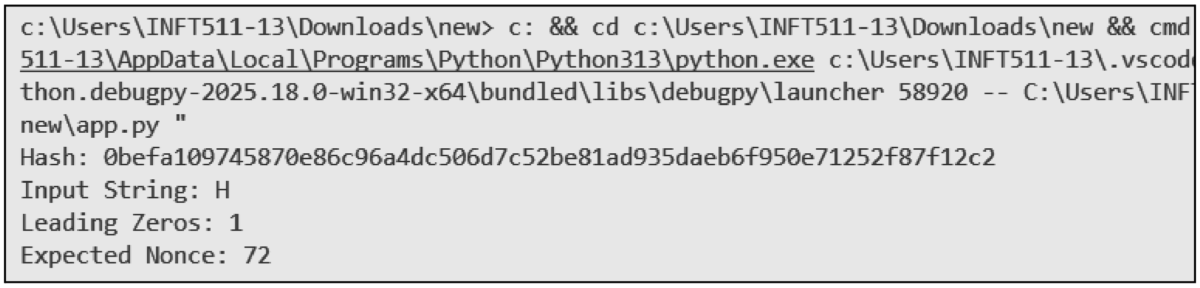


```
c:\Users\INFT511-13\Downloads\new> c: && cd c:\Users\INFT511-13\Downloads\new && cmd
511-13\AppData\Local\Programs\Python\Python313\python.exe c:\Users\INFT511-13\.vscod
thon.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher 59180 -- c:\Users\INF
new\app.py "
Enter a string: H
Enter the nonce: 1
Hash Code: 8bf797aa02bd65b7cbd52155f6e010be77bb3d461320dac4c74c12d749d81066
```

Program #3: Solving Cryptographic Puzzle for Leading Zeros

```
import hashlib
 def find_nonce(input_string, num_zeros):
        nonce = 0
        hash_prefix = '0' * num_zeros
        while True:
        # Concatenate the string and nonce
        hash_string = input_string + str(nonce)
        # Calculate the hash using SHA-256
        hash_object = hashlib.sha256(hash_string.encode('utf-8'))
        hash_code = hash_object.hexdigest()
        # Check if the hash code has the required number of leading zeros
        if hash_code.startswith(hash_prefix):
        print("Hash:",hash_code )
        return nonce
        nonce += 1
 # Get user input
 input_string = "A"
 num_zeros = 1
 # Find the expected nonce
 expected_nonce = find_nonce(input_string, num_zeros)
 # Print the expected nonce
 print("Input String:", input_string)
 print("Leading Zeros:", num_zeros)
 print("Expected Nonce:", expected_nonce)
```

Output Example:

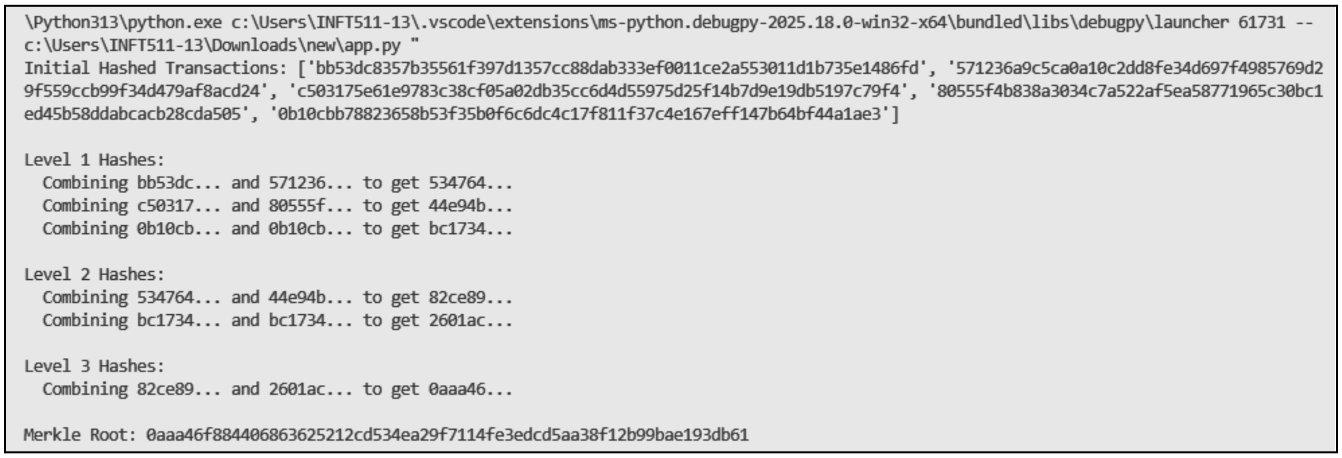


```
c:\Users\INFT511-13\Downloads\new> c: && cd c:\Users\INFT511-13\Downloads\new && cmd
511-13\AppData\Local\Programs\Python\Python313\python.exe c:\Users\INFT511-13\.vscod
thon.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher 58920 -- C:\Users\INF
new\app.py "
Hash: 0befa109745870e86c96a4dc506d7c52be81ad935daeb6f950e71252f87f12c2
Input String: H
Leading Zeros: 1
Expected Nonce: 72
```

Program #4: Generating Merkle Tree for a Given Set of Transactions

```
import hashlib
def build_merkle_tree(transactions):
    if len(transactions) == 0:
        return None
    hashed_transactions = [hashlib.sha256(t.encode('utf-8')).hexdigest() for t in transactions]
    print("Initial Hashed Transactions:", hashed_transactions)
    if len(hashed_transactions) == 1:
        return hashed_transactions[0]
    current_level = hashed_transactions
    level = 1
    while len(current_level) > 1:
        print(f"\nLevel {level} Hashes:")
        if len(current_level) % 2 != 0:
            current_level.append(current_level[-1])
        new_level = []
        for i in range(0, len(current_level), 2):
            combined = current_level[i] + current_level[i+1]
            hash_combined = hashlib.sha256(combined.encode('utf-8')).hexdigest()
            new_level.append(hash_combined)
            print(f"  Combining {current_level[i][:6]}... and {current_level[i+1][:6]}... to get
{hash_combined[:6]}...")
        current_level = new_level
        level += 1
    return current_level[0]
transactions = [
    "Alice -> Bob : $500",
    "Bob -> Dave : $340",
    "Dave -> Eve : $1000",
    "Eve -> Alice : $40",
    "Roo -> Bob : $780"]
merkle_root = build_merkle_tree(transactions)
print("\nMerkle Root:", merkle_root)
```

Output :



```
\Python313\python.exe c:\Users\INFT511-13\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher 61731 --
c:\Users\INFT511-13\Downloads\new\app.py "
Initial Hashed Transactions: ['bb53dc8357b35561f397d1357cc88dab333ef0011ce2a553011d1b735e1486fd', '571236a9c5ca0a10c2dd8fe34d697f4985769d2
9f559ccb99f34d479af8acd24', 'c503175e61e9783c38cf05a02db35cc6d4d55975d25f14b7d9e19db5197c79f4', '80555f4b838a3034c7a522af5ea58771965c30bc1
ed45b58ddabcacb28cda505', '0b10cbb78823658b53f35b0f6c6dc4c17f811f37c4e167eff147b64bf44a1ae3']

Level 1 Hashes:
  Combining bb53dc... and 571236... to get 534764...
  Combining c50317... and 80555f... to get 44e94b...
  Combining 0b10cb... and 0b10cb... to get bc1734...

Level 2 Hashes:
  Combining 534764... and 44e94b... to get 82ce89...
  Combining bc1734... and bc1734... to get 2601ac...

Level 3 Hashes:
  Combining 82ce89... and 2601ac... to get 0aaa46...

Merkle Root: 0aaa46f884406863625212cd534ea29f7114fe3edcd5aa38f12b99bae193db61
```

Conclusion

In this experiment, we successfully implemented Python programs to demonstrate the core cryptographic mechanisms of blockchain technology.

1. **Integrity Verification:** By using the hashlib library, we verified that SHA-256 generates unique, fixed-length signatures for data. We observed that even slight modifications to the input completely altered the resulting hash, confirming the algorithm's reliability for detecting data tampering.
2. **Mining Simulation:** We simulated the Proof-of-Work mechanism by implementing a nonce-finding algorithm. The experiment showed that finding a hash with a specific prefix (leading zeros) requires computational effort (brute force), effectively demonstrating how difficulty targets secure blockchain networks.
3. **Data Structure Efficiency:** The construction of the Merkle Tree illustrated how large sets of transactions can be summarized into a single "Merkle Root." This demonstrated the efficiency of blockchains in verifying transaction integrity without needing to process every single transaction individually.

Overall, the experiment confirms that SHA-256 hashing and Merkle Trees are essential for ensuring the **immutability, security, and efficiency** of distributed ledger systems.