CMSC498I: Real World Computer Security (Fall 2022) Assignment 2

Due Date: Wedensday October 26th at 11PM Eastern time.
Submission Instructions Submit Code and a Text file for responses on ELMS/Canvas.

In this assignment, you will learn how Merkle trees and Merkle Patricia tries work, particularly in applications for blockchains.

This assignment will require Python 3 and the scripts can be run by entering "python3 <filename>.py". Only make changes is the designated area in the mentioned functions within these files.

The files "hw2-merkle-trees.py" and "hw2-merkle-trees.py" contain the functions you will be implementing. The files "MT_test.py" and "bMPT_test.py" contain tests that you may run to test your code. They are not comprehensive, nor representative of how the assignment will be graded. You may modify the test files as you wish as they will not be looked at for grading.

1 Warm up Questions(5 points)

For the following warm up questions, please submit your answers in a text file (.txt). Merkle trees have many applications, but in this case we are going to consider using them to store transactions.

1.1 Naive List

Suppose that you are trying to create a simple blockchain where each block has every transaction recorded in a list. If there are t transactions, each containing L bytes, how many bytes would the transaction list be and what would be required for someone to show that a transaction was contained in that list? How much work would it take someone to check such a claim?

1.2 Naive Concatenation and Hash

Now suppose you wish to create a more compact blockchain. Instead of recording each transaction in a block, you now concatenate all transactions in a block and then hash the result using SHA-256. Instead of recording all transactions within a block, just the hash is recorded.

What would be required to prove that a transaction was contained in that block? What would it take to check the claim?

2 Merkle Trees(35 points)

Recall that a Merkle tree is constructed as follows where each node has the hash if its children and || denotes concatenation.

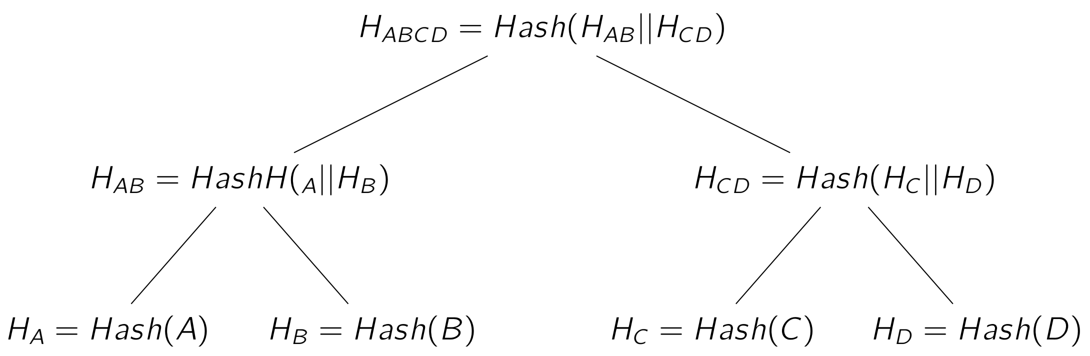
$H_{ABCD} = Hash(H_{AB}||H_{CD})$
$H_{AB} = HashH(_A||H_B)$
$H_{CD} = Hash(H_C||H_D)$
$H_A = Hash(A)$
$H_B = Hash(B)$
$H_C = Hash(C)$
$H_D = Hash(D)$

Figure 1: Example how of a Merkle Tree computes hashes.

2.1 Merkle Membership

Refer to the Merkle tree in Figure 1. Suppose a block header contains the hash of the root node, H_{ABCD}. What would be needed to prove membership for transaction C? What would a verifier compute to check that this proof is valid?

3 Implementation

The programming assignment for this section is in the file "hw2-merkle-trees.py".

The following is the class object for a Merkle Tree Node:

```
class Node:
    def __init__(self, left, right, hash_value: str,content)-> None:
        self.left: Node = left
        self.right: Node = right
        self.hash_value = hash_value
        self.content = content
```

3.1 Generating a Merkle Tree

In this problem, you will be implementing a function to construct a Merkle Tree. The input will be the transaction data in the form of an array of strings. Recall that all of the data in a Merkle tree, such as the transactions, are stored in the leaves while the other nodes are comprised of only hashes.

Implement the function "generate_tree()" that will take in a list of transactions. The function should construct a Merkle tree and return the root. There are some examples and two small test cases (function "MT_check_three_node_example()" and "MT_check_four_node_example()" in "MT_test.py"). Refer Python script for details.

3.2 Generating a Proof of Membership in a Merkle Tree

In this section, you will be implementing a function to produce the proof of a transaction. The function will take two inputs: a transaction and the Merkle tree that contains it. The output of this function should be a list of hashes that a verify will use to check that a transaction was recorded in a tree. Refer to the function "generate_proof()" for details.

3.3 Verify a Proof for Merkle Tree Membership

Given a Merkle tree and a proof of membership, implement a function to verify it. The input will be a tree, a transaction, and the proof, and the function should return True or False. Refer to the function "verify_proof()" for details.

4 (Binary) Merkle Patricia Tries(60 points)

While Merkle trees compactly record transactions, they do not support looking up data by a key. Merkle Patricia tries implement a verifiable key value store and efficiently support inserts, edits, reads, and deletes.

In this section, you will be working with binary Merkle Patricia tries. Some details will be abstracted away for simplicity. You may assume that all keys have the same length.

The exercises for this section is in the file "hw2-merkle-patricia-trie.py".

```
class MPT_Ext_Node(object):
    def __init__(self, shared_nibble, next_node=None, hash_value=None):
        self.mpt_type = MPT_Type.extension
        self.shared_nibble = shared_nibble
        self.next_node = next_node
        self.hash_value = hash_value
```

```
class MPT_Branch_Node(object):
    def __init__(self, index0=None, index1=None, hash_value=None):
        self.mpt_type = MPT_Type.branch
        self.index0 = index0
        self.index1 = index1
        self.hash_value = hash_value
```

```
class MPT_Leaf_Node(object):
    def __init__(self, key_end, value, hash_value=None):
        self.mpt_type = MPT_Type.leaf
        self.key_end = key_end
        self.value = value
        self.hash_value = hash_value
```

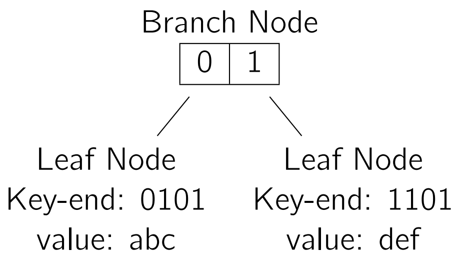

Figure 2: Example of a simple Merkle Patricia trie with 2 values. Note that the Branch node does not hold any values, only pointers to the 2 nodes. The key-value pairs are $(0101, "abc"), (1101, "def")$.

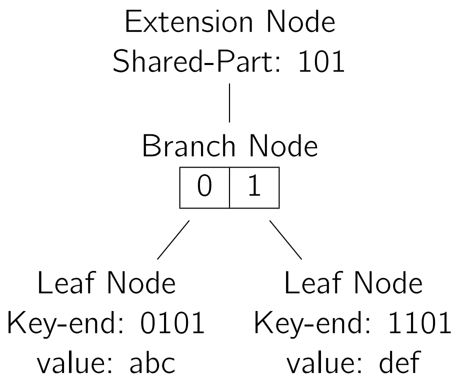

Figure 3: Example of a simple Merkle Patricia trie with all three node types and 2 values. The key-value pairs are $(1010101, "abc"), (1011101, "def")$.

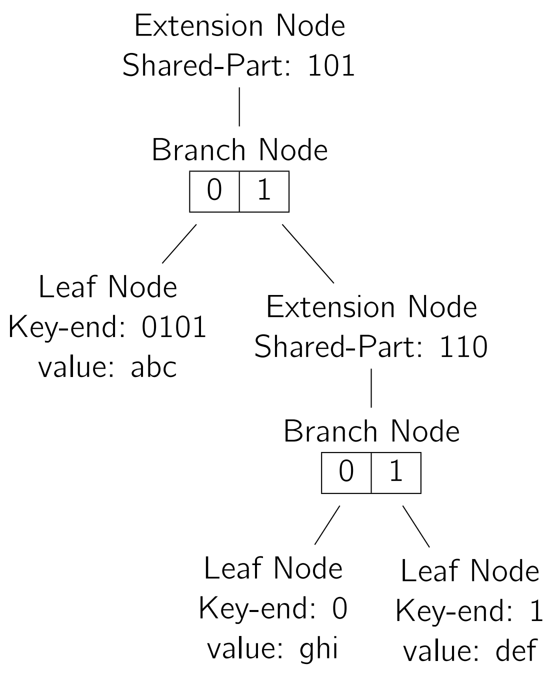

Figure 4: Example of a simple Merkle Patricia trie with all three node types and 3 values. The key-value pairs are $(1010101, "abc"), (1011101, "def"), (1011100, "ghi")$.

4.1 Insert an element into a Merkle Patricia Trie

In this section, you will adding a key-value pair to an existing Merkle Patricia trie. Refer to the function "add_to_merkle_patricia()" for details.

4.2 Edit an existing element in a Merkle Patricia Trie

Traverse a Merkle Patricia trie in order to find and edit a key-value pair within the trie. Refer to the function "edit_merkle_patricia()" for details.

4.3 Remove an element into a Merkle Patricia Trie

Traverse a Merkle Patricia trie in order to find and remove a key-value pair within the trie. Refer to the function "remove_node_merkle_patricia()" for details.

4.4 Generating a Merkle Patricia Tries

Implement a function, that given key-value pairs, will construct a Merkle Patricia trie in function "generate_merkle_patricia()". Refer to the comments for the syntax on how to build the trie.

4.5 Generating a Proof for Merkle Patricia Tries

Using the prior function to create a Merkle Patricia trie, generate a proof for one of the key-value pairs stored in it. Refer to the function "generate_merkle_patricia_proof()" for details.

There are some examples and a small test case (function "MT_check_three_node_example()" and "MT_check_four_node_example()" located in the test file, "bMPT_test.py").

4.6 Verify a Proof for Merkle Patricia Tries

Given a Merkle Patricia trie and a proof of membership, implement a function to verify it. Similiar to the Merkle tree version, the input will be a trie, a key-value pair, and the proof. Refer to the function "verify_merkle_patricia_proof()" for details.