Blockchain and Distributer Ledger – Assignment 1

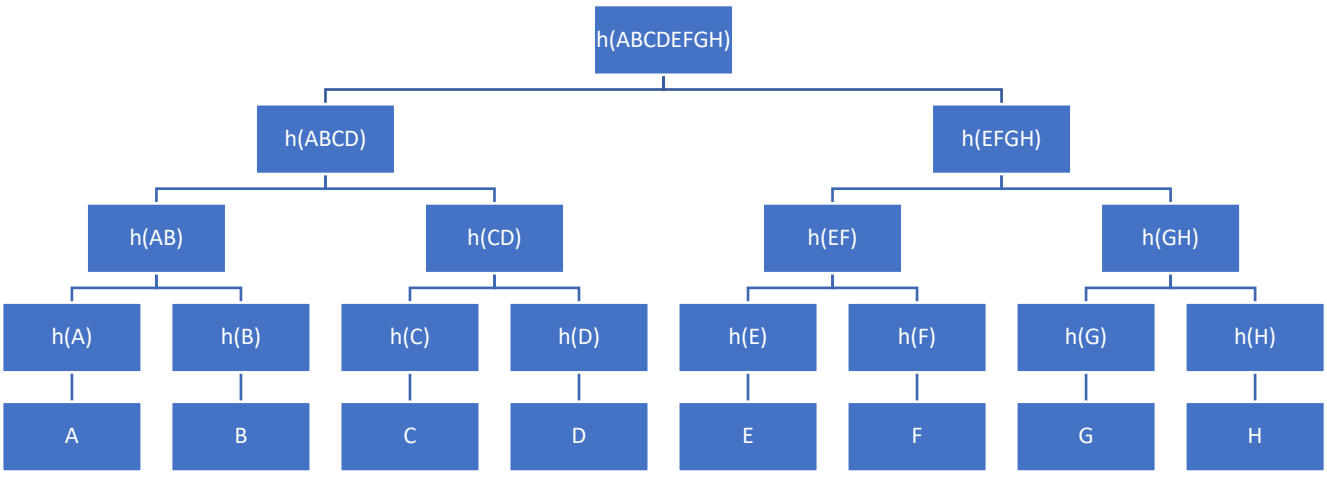
Abhirup Chakravarty – S2300928

Theoretical Questions

Q. 1.a. Formally prove (or disprove) the following proposition: "It is infeasible to forge a proof of inclusion for Merkle tree that uses a collision resistant hash function."

A. Merkle trees are build using a bottom-up approach, computing the hashes of the leaves, and using a recursive iteration approach where the hashes from the n^{th} level connections are combined to derive combined hashes for the connections at the $(n-1)^{th}$ level.

This is illustrated as follows:



Here, the hash function is denoted by h(), and A,B,C,D,E,F,G,H are the discrete data blocks hashed in the Merkle tree. The 0^{th} level node is known as the Merkle Tree Root (MTR).

Proof of inclusion is provided by giving the data block, whose proof of inclusion needs to be checked, the MTR, and the short proof of inclusion which is a set of the hashes of the sibling nodes in every level in the iterative path from the leaf-level till the MTR.

Thus, for instance, if we were to validate the proof of inclusion for block F in the previously illustrated Merkle Tree, we would need the data block F, the MTR (= h(ABCDEFGH)), and the short proof of inclusion = [h(E), h(GH), h(ABCD)].

The proof is then done as follows:

LHS = h(h(ABCD)+h(h(h(E) + h(F)) + h(GH))) [Here, "+" denotes concatenation of two hashes]
RHS = h(ABCEDFGH)

Therefore, LHS =

h(h(ABCD)+h(h(h(E) + h(F)) + h(GH)))
= h(h(ABCD)+h(h(EF) + h(GH)))
= h(h(ABCD)+h(EFGH))
= h(ABCDEFGH)
= RHS

Now say, if any of the data blocks were to be spoofed,
Let us consider that block E was altered to be E' by an attacker.
Since our basic assumption is that the hash-function is collision resistant, we can say that h(E) will not be equal to h(E') with confidence, i.e., h(E) = h(E') is a very low probability event, making it quite infeasible.

Thus, if we try to verify the proof of inclusion for the altered E' block,
We get:
LHS = h(h(ABCD)+h(h((h(E') + h(F)) + h(GH)))
RHS = h(ABCEDFGH)
The proof of inclusion in this case is the set: [h(F), h(GH), h(ABCD)]

Therefore, LHS =

h(h(ABCD)+h(h(h(E') + h(F)) + h(GH)))

= h(h(ABCD)+h(h(E'F) + h(GH)))
= h(h(ABCD)+h(E'FGH))
= h(ABCDE'FGH)
≠ RHS

Even, if the attacker alters/spoofs the short proof of inclusion for the block E', giving us [h(F)', h(GH), h(ABCD)],

It is still highly improbable to get

h(h(h(E') + h(F)')) + h(GH)) = h(EFGH),
and the same is applicable for any of the elements in the short proof of inclusion set.
Even if the attacker altered the entire short proof of inclusion, as follows:
[h(F)', h(GH)', h(ABCD)'],
It is still highly improbable to find a hash (function) that will give
h(h(ABCD)'+h(h(h(E') + h(F)') + h(GH)')) = h(ABCDEFGH)

This is primarily because the hash of the Merkle tree has remained unaltered and accommodates for (changes the hash) any change in the data passed into it as the parameter(s).

Thus, it is infeasible to forge a proof of inclusion for Merkle tree that uses a collision resistant hash function.
Q.E.D.

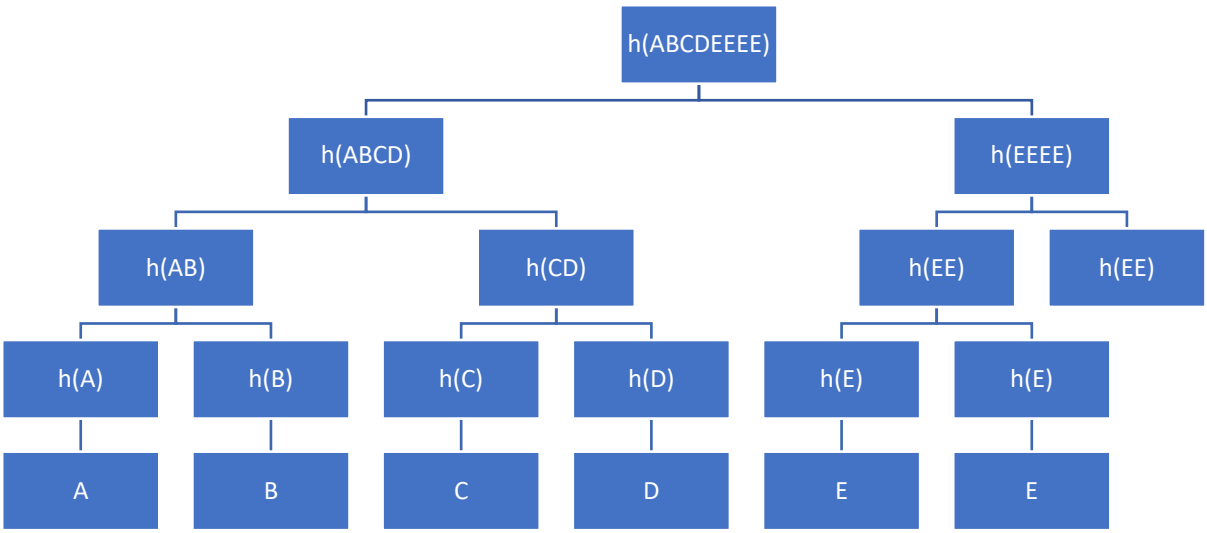
Q. 1.b. Describe (or sketch)

i) a binary full Merkle Tree,

ii) a binary complete Merkle tree,

for the following chunks of data: A,B,C,D,E. Provide the proof of inclusion for E in both cases.

A. i) Full Binary Merkle Tree:

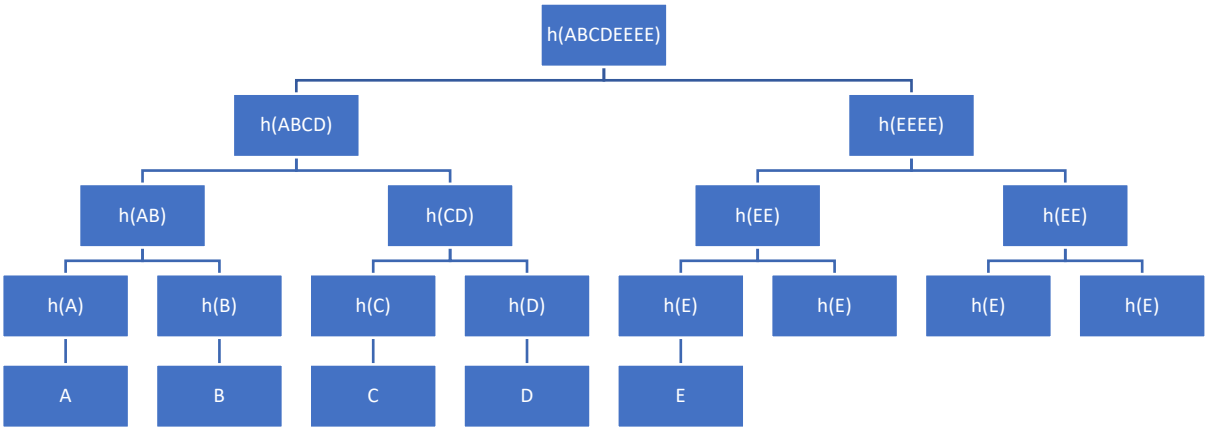


Since this is a Full Binary Merkel Tree, we can have nodes with 0 or 2 children. In that case, since we have an odd number of leaves, due to the number of data blocks being 5, we re-use the value of a node for its sibling node, wherever we would have had a single child node to a parent, otherwise. This is demonstrated as above. E is reused such that we can only have proof of inclusion for A, B, C, D and E data blocks in a sorted order.

Short proof of inclusion for E: [h(E), h(EE), h(ABCD)]

RHS
= MTR
= h(ABCDEEEE)
LHS
= h(h(ABCD) + h(h(h(E) + h(E)) + h(EE))) [Here, “+” denotes concatenation of two hashes]
= h(h(ABCD) + h(h(EE) + h(EE)))
= h(h(ABCD) + h(EEEE))
= h(ABCDEEEE)
Thus, LHS = RHS
Q.E.D.

ii) Complete Binary Merkle Tree:



Since a Complete binary Merkle tree must have all nodes above the second to last level to have 2 nodes, and all the leaf nodes must be arranged in a left-most arrangement, the above is how we can construct a complete binary Merkle tree. h(E) is reused such that we can only have proof of inclusion for A, B, C, D and E data blocks in a sorted order.

Short proof of inclusion for E: [h(E), h(EE), h(ABCD)]

RHS
= MTR
= h(ABCDEEEE)
LHS
= h(h(ABCD) + h(h(h(E) + h(E)) + h(EE))) [Here, “+” denotes concatenation of two hashes]
= h(h(ABCD) + h(h(EE) + h(EE)))
= h(h(ABCD) + h(EEEE))
= h(ABCDEEEE)
Thus, LHS = RHS
Q.E.D.

Q. 2. Derive the formula for the birthday paradox (show your work, explaining every step) and calculate the approximate number of elements needed to find a collision with at least 50% probability. Apply this to find out approximately how many Bitcoin wallets needed to initialise their seed, which is a randomly sampled sequence of 12 words from the list in https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt, to have the event that, with probability at least 50%, at least two wallets end up with exactly the same seed (calculate both with and without replacement of sampled words).

A. To derive the formula for the birthday paradox, let us consider that the number of possible dates is represented by n, and the number of people needed by k.
Since, probability of collision = 1/2 = probability of non-collision in our case, so we can compute
$P(\neg C) = 1/2$
$\Rightarrow n/n * (n-1)/n * \dots * (n-k+1)/n = 1/2$
$\Rightarrow \prod_{l=1}^{k-1}(1 - l/n) = 1/2$
since, LHS $\leq \exp(-1/n * \sum_{l=1}^{k-1}(l))$
we can approximate k by simplifying LHS to
exp(-k(k-1)/2n)),
and equating it to 1/2, as follows:
$\exp(-k(k-1)/2n)) \approx 1/2$
$\Rightarrow -k(k-1)/2n \approx \ln(1/2)$
$\Rightarrow k^2 - k - 2n * (0.693) \approx 0$
$\Rightarrow k \approx 1.177*\text{sqrt}(n)$ (Using the standard solution to the Sridharacharya equation, ignoring the negative root of k)
In case of the Bitcoin wallet subproblem, we have a pool of 2048 words to be chosen for the seed.
In case of repetitions not allowed,
$n = {}^{2048}P_{12} = 5.27 \times 10^{39}$
Thus, we can simply use this value to compute k as follows:
$k \approx 1.177 * \text{sqrt}(5.27 \times 10^{39})$
$\Rightarrow k \approx 8.546 \times 10^{19}$
In case of repetitions allowed,
A seed contains 12 words, randomly sampled, allowing repetitions. Thus, we can say that in this case, $n = (2048)^{12}$.
Thus, we can simply use this value to compute k as follows:
$k \approx 1.177 * \text{sqrt}(2048^{12})$
$\Rightarrow k \approx 8.685 \times 10^{19}$

Q. 3. Assume that all Bitcoin miners are honest except one (the attacker). A miner creates a block B which contains address α, on which they want to receive their rewards. The attacker changes the contents of B, replacing α with an adversarial address α'. What needs to happen for the attacker to receive the rewards for B?

A. There are two events that can happen, the first one being extremely improbable, and second one being slightly impractical. When the malicious miner alters the address, it is highly probable that the hash of the block changes and all the other nodes (majority) detects the change and reject the block to be accepted into the majority chain, thus preventing corruption. But if the address was curated in a way that it wouldn't alter the hash code (the hash function's collision-resistance fails), although highly unlikely, it will be validated by the network. The more probable, but slightly impractical way for the attacker to receive the reward at the address α' is for the adversary to attempt a 51% attack, where it controls at least 51% (majority) of the hash rate of the network. Thus, the attacker ensures that it controls what blocks the blockchain validates and what blocks it rejects, as in accordance with the majority consensus rule. This is impractical because of the vested running costs required to obtain such majority in the network at the present distributed nature of the blockchain. However, it is important to note that several such attacks have been recorded as late as 2021 on several bitcoin forks.

Q. 4. Give two detailed examples of how an orphan block can be created in Bitcoin.

A. Let's assume that two miners A and B has mine the same block at nearly the same time. Due to latency in broadcasting, or different internet bandwidth, A has broadcasted it to their neighboring nodes, and with some delay, B has broadcasted it to their neighboring nodes. At this point, there is a certain fork in the blockchain, as both the blocks are accepted in two different groups of the network. Eventually, another miner C, who has accepted the block mined by A to be true, mines another block, and adds it to the chain, and publishes it. Because the fork with A's block has a higher proof of work (majority chain), it is accepted as the true chain, and the block mined by B is orphaned. However, it doesn't necessitate that it will resolve only by another addition of block and may go on to create forked chains of any number of blocks, but eventually one will become longer, and will be accepted as the true chain.

Another interesting example could be seen in case of double spending with a 51% attack. If an attacker had most of the hashing power (more than 50%) of a network, they would have a higher probability of finding the next block. The idea is that instead of broadcasting their block immediately, they do it around the same time any block is announced in the main chain. Here, the idea is that they need to be very precise (timewise) and lucky (proof of work, wise) to just stay ahead of the actual chain. If they had a transaction recorded in the main chain, but didn't record it in their own chain, after a few confirmations, there would be two competing chains. Once the attacker has concluded all their transactions, they mine the next block and broadcast it immediately with an altered history (without their transactions, that were recorded in the actual chain). Because the attacker has the chain that has the most number of blocks, the majority chain rule and proof of work protocols dictate that the network will identify it as the new main chain and broadcast it across the network. As a result, the attacker would have nullified their transactions, altered the history on the blockchain, and achieved double spending. As a direct result of this, the actual main chain (with proper history) gets rejected and the blocks become orphaned.

Q. 5. Construct a digital signature scheme, based on a hash function, that is one-time secure (i.e., it is secure if each private key signs only a single message). Describe in detail how the keys are generated and how signatures are issued and verified.

A. Let us consider using two collision-resistant hash functions H_1 and H_2 with outputs in 256 bits.
The signer should generate 512 random numbers, each 256 bits, thus giving 512 x 256 = 128 Kbits. This will be the sender's private key.
The public key is the set of hash images of all the numbers present in the set of random numbers generated by the signer (128 Kbits in size), obtained using the function H_1.
To sign the message, the signer hashes the message using H_2 and reads the message in bit-wise representation. Pseudocode for signing the message is as follows:

Algorithm 1: Computes the Signature for given message and private key.

```
1  Compute_Signature (message, private_keys[1...512])
   Input  : The String message is what the signer wants to sign for, and
            the Set private_keys[1...512] is the set of the randomly
            generated 512 numbers owned by the signer.
   Output: The Set signature[1...256] based on the private keys and
            message as provided by the signer, initialised to be an empty
            state at the beginning of the process.
2  Initialize idx to 0. H_2 is the collision resistant hashing algorithm used
     to hash the message to convert it into a 256-bit sized data chunk. for
     i = 1 to 256 do
3  |  if H_2(message)[i] == 0 then
4  |  |  signature.push(private_keys[idx]);
5  |  else
6  |  |  signature.push(private_keys[idx+1]);
7  |  end
8  |  idx = idx + 2;
9  end
10 return signature[1...256];
```

To verify, the following pseudocode is used:

Algorithm 2: Verifies the signature for given message and public key.

```
1  Verify_Signature (message, public_keys[1...512], signature[1...256])
   Input  : The String message is what the verifier wants to verify as
            authentic. The Set public_keys[1...512] is the set of the
            public keys broadcasted by the signer that the verifier must
            use to check the authenticity. The Set signature[1_256]) is the
            signature for the respective message received by the verifier.
   Output: Boolean, True signifying verified.
2  Initialize idx to 0. Initialise verification_set[1...256] to empty set. H_2
     is the collision resistant hashing algorithm used to hash the message to
     convert it into a 256-bit sized data chunk. for i = 1 to 256 do
3  |  if H_2(message)[i] == 0 then
4  |  |  verification_set.push(public_keys[idx]);
5  |  else
6  |  |  verification_set.push(public_keys[idx+1]);
7  |  end
8  |  idx = idx + 2;
9  end
10 if verification_set.equals(signature) then
11 |  return True;
12 else
13 |  return False;
14 end
```

Hands-on Questions

Q. 6. Using the course's Ethereum testnet, send 1 ETH to the address of a fellow student. Write a small description on how you conducted the payment, including the transaction's id and addresses which you used.

A. The steps involved in the conduction of the payment are as follows:

Step 1. Log in to your Metamask Wallet UI using your password.

Step 2. Verify that you have the necessary funds to make the transaction, calculated as: Amount needed to be transferred + estimated required gas.

Step 3. Click Send and enter recipient's address in the required input text field. Enter the amount of ETH to be transferred. Check, and if required, update the Gas Price and Gas Limit. In this case, default Gas Price: 1 Gwei (1 Giga-wei = 10^9 wei) and default Gas Limit: 21000 were used.

Step 4. Click Next. Verify Estimated Gas Fees and Total Fees (Amount + Gas Fee). Click Confirm.

Step 5. In activity list, see that the transaction is registered and the status shows Pending. Wait.

Step 6. Check that the status of the transaction is now Confirmed.

Transaction Id:
0x65a1470a26099a3c7bf5f812874b060c37aad7259215cf158505bab789ec1116
Sender's Address: 0xA06E1629d640230167678ddBfD817E2ad4E706A8
Recipient's Address: 0x2306CbbEA023A92316e0Ee0e054E08aB69747e04

Q. 7. A smart contract has been deployed on the course's testnet. Its code is available here (https://github.com/Blockchain-Technology-Lab/bdl-course/blob/master/Bank.sol) and its deployed address is: *0xA8D9D864dA941DdB53cAed4CeB8C8Bcf53aFe580* You can compile and interact with it using Remix (https://remix.ethereum.org). You should successfully create a transaction that interacts with the contract, either depositing to or withdrawing from it some coins. Describe the contract's functionality (that is, the purpose of each variable, function, and event) and provide the id of the transaction you performed and the address you used.

A.

Transaction Id:
0x7b1439e3d2a8cca51a29c3f578f57330db2ab9b722171588e695a7cdd07bdadf
Sender's Address: 0xA06E1629d640230167678ddBfD817E2ad4E706A8
Recipient's (Contract) Address:
0xA8D9D864dA941DdB53cAed4CeB8C8Bcf53aFe580

The following line uses the directive pragma which dictates which versions of solidity the compiler can/will use for compiling the source code:

```
pragma solidity >=0.7.0 <0.9.0;
```

The following statement starts the definition of the contract (equivalent to a class in standard object-oriented programming nomenclature) Bank:

```
contract Bank {
```

The following defines a map named balance that maps key type address to value type unsigned integers of 256bits:

```
    mapping(address => uint256) balance;
```

The following defines a dynamic array named customers which stores the value type address:

```
    address[] public customers;
```

An event is a contract component that can be inherited. When it is emitted, and the arguments passed are recorded in transaction logs, which again can be saved on the blockchain to which the contract belongs. They can be accessed using the contract's address if the contract is present on the blockchain. An event generated is however not accessible from within contracts, including those that originated and emitted it. Deposit is such an event available to the Ethereum blockchain, which takes the parameters of the sender's address and a message string as input and logs these on the blockchain when the contract is called, and this method is emitted. The following implements the event Deposit from:

```
    event Deposit(address customer, string message);
```

Similarly, Withdrawal is such an event available to the Ethereum blockchain, which takes the parameters of the sender's address as input and logs it on the

blockchain when the contract is called, and this method is emitted. The following implements the event Withdrawal from:

```
event Withdrawal(address customer);
```

The following defines a function deposit within the contract Bank that extends payable which allows the function to receive Ether. It is public, meaning it can be accessed from anywhere (it is an open access classifier). Otherwise, the transaction would fail. It takes discrete input of a string parameter named message that is stored in memory (as described by the memory keyword). The require method checks whether the value attribute of the object msg is greater than 10. In practicality, it checks if the deposited is more than 10 wei. In the next line, it deducts 10 wei from the deposit made (as a fee, I suppose) and refers to the balance stored in the balance array, indexed by the sender's address in the map, and adds the value sent by the sender, after the said 10 wei subtraction. The sender's address is then pushed into the customers address dynamic array. The deposit event is then emitted with the sender's address and the discrete input message (that can be passed using the remix UI in our case, or through API calls in a proper deployment/integration with some application).

```
function deposit(string memory message) public payable {
    require(msg.value > 10);
    balance[msg.sender] += msg.value - 10;

    customers.push(msg.sender);

    emit Deposit(msg.sender, message);
}
```

The following defines a function withdraw within the contract Bank that is public, and thus can be called or accessed anywhere. The first line retrieves the balance of the person who called the method by indexing their address's data in the map balance, and stores it in an unsigned integer (256-bit) parameter named b. Then their balance in the balance map is emptied, by saving it as 0. Then the address of the caller (which is presently in address type) is casted to payable type and the balance of b (in wei) is transferred to them using the implicit transfer function of the payable type. The withdrawal event is emitted with the address of the caller.

```
function withdraw() public {
    uint256 b = balance[msg.sender];
    balance[msg.sender] = 0;
    payable(msg.sender).transfer(b);

    emit Withdrawal(msg.sender);
}
```

The following defines a function getBalance within the contract Bank that is public, and thus can be called or accessed anywhere. Since it only reads but doesn't alter the state variables defined in the contract, view keyword is used. It returns the type uint256 (unsigned integer of 256 bits). It directly refers to the value stored in the balance map using the caller's address and thus effectively just returns their balance (as uint256).

```
    function getBalance() public view returns (uint256) {
        return balance[msg.sender];
    }
```

(Can be the predicament of a classic rug pull joke) The following function empty is also a public function like all the other above. It checks if the address of the caller is the same as the first ever address pushed to the customers dynamic array (ideally the address of the owner of the person/body deploying the code but can be the address of first ever customer of the bank). If the check passes, then the address is casted to payable type and the entirety of the balance owned by the contract referred by "address(this).balance" is then transferred to the payable address. The this keyword refers to the contract object, and ".balance" refers to the balance attribute of the object. "address(this)" casts the instance of the object (contract) bank to the type address, which essentially owns the property balance.

```
    function empty() public {
        require(msg.sender == customers[0]);

        payable(msg.sender).transfer(address(this).balance);
    }
```

The following line concludes the definition of the contract bank.

```
}
```