Part 1: Theory (5 x 12 points)

1. How are hash functions used in a Merkle Tree?

Answer:

Hash function is one kind of function that could be used to realize a mapping between inputs and outputs. There are no limitations on the length and type of the input data, and they would be translated to certain length of data namely “hash value” (it depends on the adopted algorithm e.g., SHA-256 create 32bytes data).

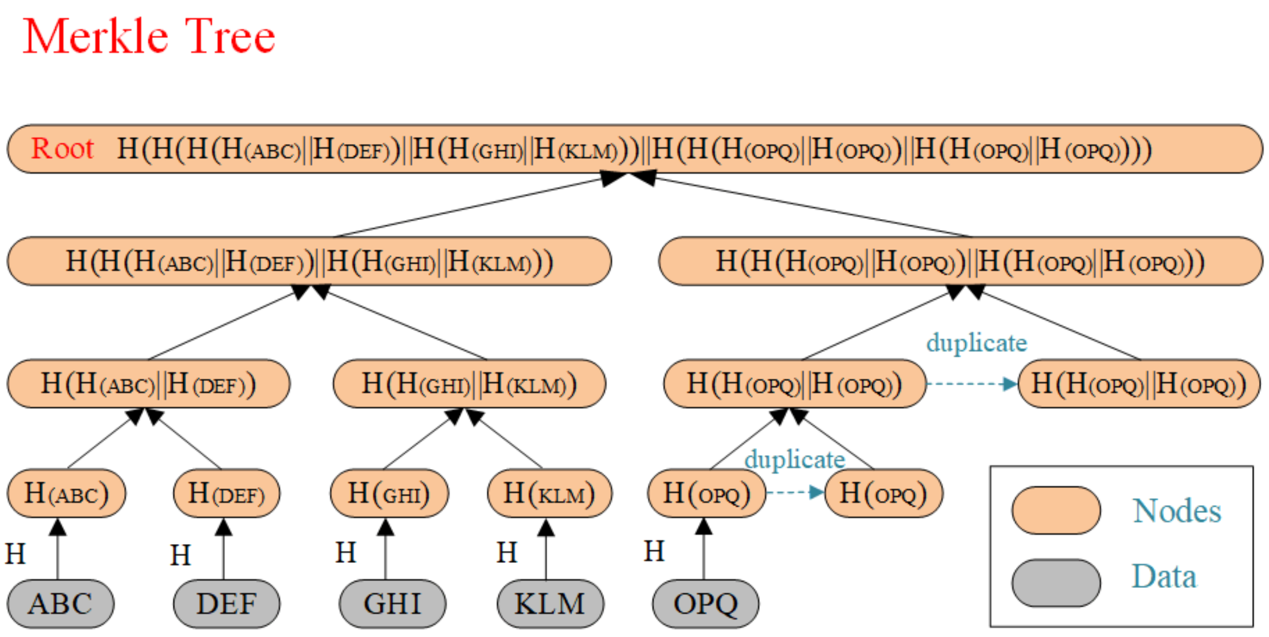
In Merkle Tree, the usage of hash function could be reflected in two aspects:

Firstly, the construction of a Merkle tree, the raw data could be divided into several chunks, and each chunk would be imported into hash function, and its hash value would be used as the node value to create a leaf node. The parent note would also be created with hash functions where the concatenation of the hash values from its leaf nodes would be imported as inputs. Trough this way, step by step, we finally get a top hash value which would be the root of this Merkle Tree.

Secondly, hash function could be used to verify whether one chunk of data is included into some file or not. For instance, we only need to calculate its own hash value, and use the hash value of its connecting leaf node to calculate the hash value of their parents nodes, and though the iteration of this step, we can finally get the root hash basing on this chunk of data, and we can simply compare this hash value with the root hash of the file, if they are equal, it means that this chunk of data is included.

1.a. Describe (or sketch) how a (binary complete) Merkle Tree is constructed for the following 5 chunks of data: ABC, DEF, GHI, KLM, OPQ

Answer to sub-question a:



Merkle Tree
Root H(H(H(H(ABC)||H(DEF))||H(H(GHI)||H(KLM)))||H(H(H(OPQ)||H(OPQ))||H(H(OPQ)||H(OPQ))))
H(H(H(ABC)||H(DEF))||H(H(GHI)||H(KLM)))
H(H(H(OPQ)||H(OPQ))||H(H(OPQ)||H(OPQ)))
H(H(ABC)||H(DEF))
H(H(GHI)||H(KLM))
H(H(OPQ)||H(OPQ))
duplicate
H(H(OPQ)||H(OPQ))
H(ABC)
H(DEF)
H(GHI)
H(KLM)
H(OPQ)
duplicate
H(OPQ)
H
H
H
H
H
ABC
DEF
GHI
KLM
OPQ
Nodes
Data

Fig1. The construction of binary complete Merkle Tree with limited data

1.b. Describe (or sketch) how a Patricia Trie is constructed for the following key/valuestore: {bla: 17, blabla: 28, bored: 53, board: 39, aboard: 42, abroad: 17}

Answer to sub-question b:

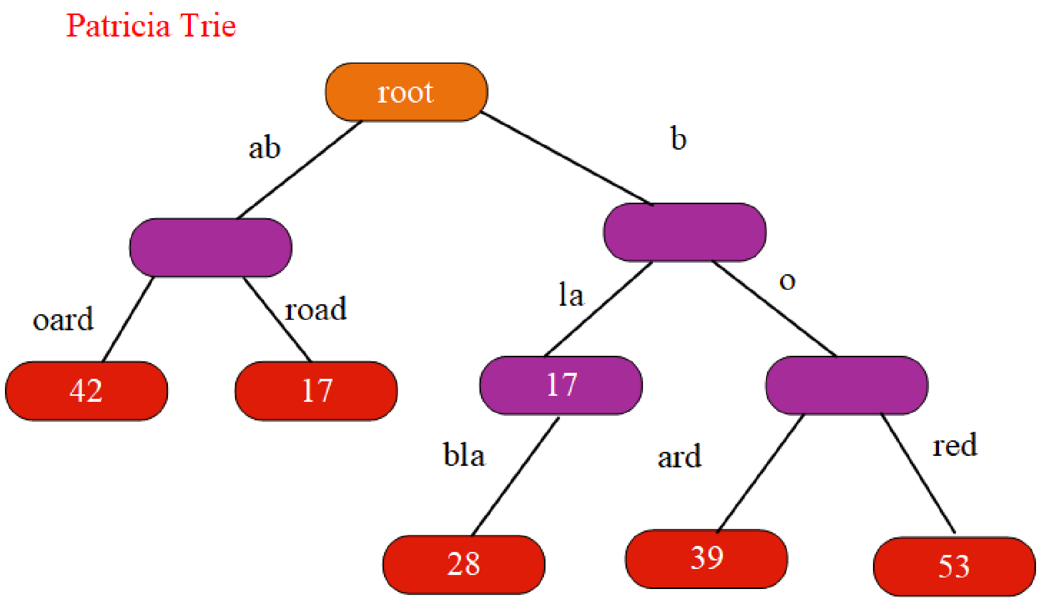


Fig2. The construction of binary complete Merkle Tree with limited data

2. Derive the formula for the birthday paradox (show your work, explaining every step) and calculate the number of elements needed to find a collision with at least 50%.

Answer:

For instance, there are *n* possible dates and *k* people. To avoid collision, the first person has *n* choices from *n* dates, and the next one has *n*-1 choices from *n* dates, and step by step, the kth person has *n-(k-1)* choices. So, the possibility is:

$$P(\neg Collision) = \frac{n}{n} \times \frac{n-1}{n} \times \frac{n-2}{n} \times \frac{n-3}{n} \ldots \frac{n-(k-1)}{n} = \prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right)$$

As we all now the Taylor expansion of e^x is:

$$e^x = 1 + x + \frac{x^2}{2!} + \ldots$$

Let's ignore its high order term, when $|x| < 1$ we can get the expression that:

$$e^x \approx 1 + x$$

Let's replace x with $-\frac{i}{n}$:

$$P(\neg Collision) = \prod_{i=1}^{k-1} e^{-\frac{i}{n}}$$

Furtherly:

$$P(\neg Collision) = e^{-\left[\frac{1}{n}+\frac{2}{n}+\ldots\frac{k-1}{n}\right]} = e^{-\frac{k\times(k-1)}{2n}}$$

Since we assume that $P(\neg Collision) \leq \frac{1}{2}$,it means that:

$$-\frac{k(k-1)}{2n} \leq \ln\left(\frac{1}{2}\right)$$

$$\frac{k(k-1)}{2n} \geq -\ln\left(\frac{1}{2}\right)$$

$$k(k-1) \geq 0.6931 \times 2 \times n$$

$$k \geq 1.177\sqrt{n}$$

Apply this to find out how many Bitcoin users are needed to initialize their wallet's seed, which is based on a random selection of 12 random words from the list, to have the event that, with probability at least 50%, at least two users end up with exactly the same seed.

Answer:

To solve the question with the conclusion we draw from the birthday paradox, the first step is to obtain the range of elements to be selected. After investigation, it could be found that the order of words is decisive to the generation of seed, and the selection of 12 words allow duplication, in other words, users are allowed to choose one specific word more than once, so the possible selection range of *n* would be:

$$2048^{12}$$

Let's express it into the formula we derived in last question:

$$k \geq 1.177 \times 2048^{6}$$

It means that:

$$k \geq 1.177 \times 2^{66}$$

In conclusion, those large number of Bitcoin users are needed to have the event that, with probability at least 50%, at least two users end up with the same seed.

3. A miner creates a block B which contains address α, on which he wants to receive his rewards. An attacker changes the contents of B, such that instead of α it defines a new address α', which is controlled by the attacker. Will the attacker receive the rewards that the miner tries to claim and why (or why not)? Give a detailed explanation of your answer.

Answer:

No, the attacker would not get the rewards that the miner tries to claim because the attacker modified the address from α to α' thus the transaction hash has been changed in which case the Merkle hash would also be different with block B. So, the attacker has to calculate a new Merkle hash based on the transaction with α' as address. However, after he calculates a new Merkle hash, the block hash would also be different, which would not satisfy the POW. When the attacker tries to broadcast this block, his POW could not be verified, so he could not get the rewards that belongs to block B.

The only thing that he can do is to calculate a new block hash to proof his work. This procedure would take lots of time, and its difficulty is equal to create a new block. At the meanwhile, other miners may continue to work on the chain based on block B and try to create new blocks after that. With more and more blocks are created behind block B, the transactions in block B would be more and more valid by the subsequent blocks. In this case, any attempts aimed at rewriting the transaction would be less and less possible.

However, if the attacker controls most of the hash rate in the community and based on his modification creating a new block B' that satisfy POW, he is possible to get his rewards. However, in this case, his behavior could not be regarded as an "attack", things would change to a competition between block B and B' both of which wants to be included into the longest chain. If the block B' wins finally in this competition, he does be able to get his rewards, but in this case, he deserves it.

4.Give a detailed example of how an orphan block can be created in Bitcoin

Answer:

For example, two miners may respectively find a valid block at the same time or in a similar time. If they both meet the requirement of difficulty, these two blocks are simultaneously valid. Typically, other miners would accept the valid block they first receive and try to obtain next block based on this block. Because the broadcasting sequence in P2P is uncertain, so presently there would exist two valid blocks among the community. This phenomenon is named "Branch". In this case, currently there would exist more than one branch and miners would also be divided into more than one part and each of them would work on the branch they choose.

However, with the continuous calculation, new blocks would be constantly created and there would exist one time that one branch of the block chain would be the longer than another one or would be the longest one in branches. According to the consensus algorithm, the longest chain would be regarded as the main chain and all the miners would transfer to this branch and work on this chain. In this case, the blocks which used to be in the longest chain but finally lose the game in the competition would be discarded as the "orphan" block and the miner who created this block could not get his reward from creating this block.

5. A signature scheme is EU-CMA secure if, for every probabilistic polynomial-bounded adversary A, A cannot win the EU-CMA security game. Write a brief description of the EU-CMA game.

Answer:

If we try to define a game between adversary and the signature scheme:

1. We have a key generation algorithm **KeyGen** that would return sign key (*sk*) and verification key (*vk*)
2. An adversary A has the access to *vk* as it is public to everyone
3. A creates message *m*
4. A sends it to Challenger and asks for Challenger's signature
5. Challenger creates a signature with **Sign** algorithm, **Sign**(*m*,*sk*)
6. A gets the signature s from Challenger
7. A repeats step 3 to steps 6

8. A outputs another signature s' based on the message he chooses *m'*
9. A wins the game if (*m',s'*) get verified by the **Verify** algorithm
10. If in the polynomial-bounded times, A could not win the game, it satisfies the EU-CMA security

Then, consider the following example: Assume a digital signature scheme Σ = (KeyGen, Sign, Verify) with security parameter λ (256 bits). Alice generates a pair of verification/signing keys (vk, sk) via KeyGen and sends vk to Mallory and Bob. Then, consider the following scenarios:
a. Mallory sends Alice a message m and requests a signature. Alice runs Sign (m, sk) responds with (m, σ, vk). Then, Mallory sends Bob (m', σ', vk), where m' ≠ m and σ' ≠ σ, and Bob accepts it as validly signed. For each case, explain in detail whether, based only on the described scenario, Mallory wins the EU-CMA game. In both cases (if yes or if no), can we conclude whether Σ satisfies the EU-CMA property? If not, why not? If yes, how is this proven, given only the described scenario? If we cannot conclude whether it is secure, explain what information is missing.

Answer to a:

Mallory wins the EU-CMA game as he is capable to create a new message-signature pair (*m', σ'*) which is different to the message based on which he gets a signature from Alice and this new message-signature pair (*m', σ'*) gets verified by Bob with verification key *vk*.

Since Mallory is capable to win the game, we can conclude that the first digital signature scheme Σ = (KeyGen, Sign, Verify) could not satisfy the EU-CMA property. Because basing on the given described scenario we can find that Mallory wins the game in his first attempt, in other words, in the polynomial-bounded times, he is able to win the game. According to the definition of EU-CMA security, this scheme could not satisfy the EU-CMA properties.

b. Mallory sends Alice two messages m, m', where m' ≠ m, and requests a signature for each. Alice runs Sign (m, sk) and Sign(m', sk) and responds with (m, σ, vk) and (m', σ', vk) respectively. Then, Mallory sends Bob (m', σ', vk) and Bob accepts it as validly signed. For each case, explain in detail whether, based only on the described scenario, Mallory wins the EU-CMA game. In both cases (if yes or if no), can we conclude whether Σ satisfies the EU-CMA property? If not, why not? If yes, how is this proven, given only the described scenario? If we cannot conclude whether it is secure, explain what information is missing.

Answer to b:

Mallory has not won the game yet because he just transmitted a signature from Alice to Bob. This signature has been signed by Alice, and everyone can check it with the verification key *vk*. Since Alice did sign to this message, this signature pair is supposed to pass the verification.

However, till now we could not conclude whether this scheme satisfies the EU-CMA property or not, because we are not aware of the polynomial-bound of this scheme. One time of attack could not prove whether this adversary is able to win the game or not. The adversary is allowed to send as many messages as he wants in the polynomial-time. In this case, further information about whether other messages are validated or not would be needed to analyze whether this scheme satisfies EU-CMA property or not.

Part 2: Hands-on (10 + 30 points)

1.Using the course's private Ethereum chain, 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. (Instructions on how to connect to the course's private chain are available here)

Answer:

To send 1 ETH to other students, there are several steps: Firstly, press the "send" button on the webpage of my account, then enter the address of the receiver, in this case it's an address from my classmates: "0x8df84aE613D0BC2623D383dA47a4ccf45a085f82". After that I was requested to enter the amount of Ether to be send, and there exist the options for Gas price and Gas limit. Let's leave it be the default values, and press confirm button. A confirmation message appeared, and now I can check the details of this transaction in the activity page. There shows that here exists one deal. And the transactions id is "0x570ca66360780fcbba8320c4c1ff240081e045517df70a4bd94388b80667193a"

2.A smart contract has been deployed on the course's private chain. You may find its code here and its deployed address is: 0xA8D9D864dA941DdB53cAed4CeB8C8Bcf53aFe580. You can compile and interact with it using Remix. You should successfully create a transaction that interacts with the contract, either depositing to or withdrawing from it some coins. Describe briefly 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.

Answer:

```
contract Bank {
    mapping(address => uint256) balance;
    address[] public customers;

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

    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);
    }

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

        emit Withdrawal(msg.sender);
    }

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

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

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

Fields of the Contract:

1. *balance*: A HashMap map address (Type: address) to their balance (Type: integer). This variable could be used to store the address and balance of customer.
2. *customers*: A container that contain customer's address (Type: address). The "public" keyword implies its scope which means that this variable is visible and could be checked by everyone in the Ethereum community.
3. **event** *Deposit*: When Deposit event was emitted, it would record the Customer's address and the message he/she left into the transaction's log.
4. **event** *Withdrawal*: When Withdrawal event was emitted, it would store the Customer's address into the transaction's log.

Functions of the Contract:

1. **deposit (string memory message)** public payable:
   This function takes one string typed formal parameter as its inputs, it is just like in the real world when we are depositing money into our account, you can choose to leave a message. The keyword "memory" claims that its argument is stored into EVM's memory. "public" means that everyone can call this function. Besides, it has a modifier "payable", a reserved keyword in solidty which

means that this function can receive Ether. In its function body, it can be found that there is a condition statement which requires that only if the value of the message sender is over 10, this function could be executed. It is worth to mention that msg.sender represents the person who currently calling the function and msg.value is the number of Wei sent by this message-sender. The values of all members of msg, including msg.sender and msg.value can change for every external function call. If the value is less than 10 Wei, this call would fail, and the message-sender would be charged with the execution fee. When the requirement is satisfied, it would push the value of the sender into his/her balance account, but it would take 10 Wei as fee. Besides, it would record the address of the sender as a customer. Finally, at the end of this function body, it would emit the *Deposit* event and pass the parameter of sender's address and message to this event. Generally, it realizes the function that when it is called by some user, it would push the money sent from the user and deposit it into his/her balance, but it would take a hand fee of 10 Wei. The amount of money to be deposit is dependent on the value the user sent.

2. **withdraw ()** public:
   The header of this function implies that its scope it public, which means that its open for everyone to call. Besides, this function does not need a parameter as inputs. In its function body, it firstly declares a local variable *b* to contain the value of the balance which belongs to the message-sender. Then it sets the balance of this account to 0. After that, it firstly uses a type conversion, to make the address of the message-sender payable, and transfer *b* Ether back to the sender. After that it would also emit an event named Withdrawal. Generally, it realizes the function that when it is called by some user, it would withdraw all the money the user deposited into his/her balance.
3. **getBalance ()** public view returns (uint256):
   The declaration of this function claims that it also does not need a parameter as inputs, but this function has a "view" keyword, which implies that it would not cause any fee to call it. Besides, it also means that this function would not change any fields in this contract. The "returns" keyword and "unit256" in the bracket claims that it would return a number and its data type is unsigned 256 bits integer. In the function body, the function uses the senders' address as a key to get its balance from the HashMap balance [] and return it. Generally, this function allows the user to check their balance.
4. **empty ()** public:
   This function does not need any arguments and could be called by everyone. In its function body, through the sentenced in line 30, it can be found that only the first customer of this contract is allowed to call this function. The next sentence indicates that this function would transfer all the Ether remained in the balance of this contract to the first customer's address.

ID of transaction and address:

1. I deposited 1 ETH to the contract, the transaction ID is listed below:
   "0x7741372b348d5f1b5e02c5e147a8055f80afe45eb69e96c6cf52de6994e9e001"
2. The address that I used to perform this transfer is listed here:
   "0x83646832DcD819f285B95DCfe1f85942037337F4"

3. The address of the deployed contract is:
   “0xA8D9D864dA941DdB53cAed4CeB8C8Bcf53aFe580”