24-Sep-2021

CS77: Introduction to Blockchain and Cryptocurrency

Mid-Semester Exam.

Name: M. Maheeth Reddy

Roll No.: 1801CS31

Answers

Ans1: Reason why Merkle Tree is used to store bitcoin transactions in a block:

Merkle tree maintains the integrity of the data. If any single detail of transactions or order of the transactions changes, then these changes are reflected in the hash of that transaction. This change would cascade up the Merkle Tree to the Merkle Root, changing the value of the Merkle Root and thus invalidating the block. So, everyone can see that Merkle Tree allows for a quick and simple test of whether a specific transaction is included in the set or not.

Merkle Trees have the following benefits:

1. It provides a means to maintain the integrity and validity of data.
2. It helps in saving the memory or disk space as the proofs, computationally easy and fast.
3. Their proofs and management require tiny amounts of information to be transmitted across networks.
4. Membership of its items can be verified in $O(\log n)$ without downloading an entire block [SPV- Simplified Payment Verification]

Preventing Double Spending Attack:

Consider the following scenario:

Alice has some bitcoin with her. She has a malicious intent and is trying to spend that money twice in two seperate transactions by sending the same bitcoins to Bob and Charles. For simplicity, I will call the transactions with Bob & Charles as T_1 & T_2 respectively.

Now, T_1 and T_2 will go into the pool of unconfirmed transactions. If T_1 is confirmed first, it is added to the subsequent block and T_2 is would become invalid.

Similar thing would happen if T_2 is confirmed first. Then T_1 would be invalid. In case T_1 and T_2 are confirmed simultaneously, the transaction with the highest number of confirmations will be included in the blockchain, and the other one will be discarded.

Here, we can notice an issue. Assuming T_1 was confirmed & T_2 failed, Charles (the recipient in T_2) would not receive his anticipated bitcoins though does not have a part in the failing of T_2. So, users are advised to wait for 6 more subsequent blocks being added to the bitcoin blockchain after their transaction has been added to a block. Once, that is done, users can safely assume that the transaction is valid.

Double Spending Attacks could also occur when an attacker can control atleast 51% of the network's nodes. Then they could attempt to reverse these attacks and create a seperate, private blockchain. However, the rapid growth of bitcoin network and its Proof of Work (PoW) consensus model requires an enormous amount of computing power, to take over the network & reverse the transactions. So, the 51% attack is impossible in reality.

So, by reaching consensus through Proof of Work mining, and combining complementary security features of the blockchain network and its decentralized network of miners to verify the transactions before they are added to the blockchain, we can prevent the double spending attacks on Bitcoin blockchain.

Ans 2: We are given 2 valid transactions T_1, T_2 and a new transaction T_3. For the i^{th} transaction, the hashes are denoted by h_i, public Key as P_i, private key as S_i

Each transaction consists of 3 parts:

① Metadata - contains information related to version,
no. of input,
no. of output,
size etc.

② Inputs - contains hash of previous transaction,
index of previous transaction's output that's being claimed
signature of the claimer

③ Outputs - contains value that is being transferred,
script containing hash of public key of the recipient

The inputs and outputs both form an array and contain scripts.

To validate a new transaction, we combine its input script & the earlier transaction's output script. The Bitcoin scripting language checks validity of the transaction using stack-based execution. So, to check T_3's validity we have to consider the given transaction:

<u>Metadata</u> - "hash" : "h3", ← T_3 hash (unique)
"ver" : 1, ← software version
"vin_sz" : 2, ← no. of inputs
"vout_sz" : 1, ← no. of outputs
"lock_time" : 0,
"size" : 604, ← size of block

P.T.O

<u>Inputs</u>: "in": {
{
"prev_out": {
"hash": "h1",
"n" : 0
}, "scriptSig": "S2 p2"
},
{
"prev_out": {
"hash": "h2",
"n" : 0,
}, "scriptSig": "S2 p2"
}
}

In T_3, we have 2 inputs, the first input should refer to h_1 i.e., the hash pointer to T_1. The index of T_1's output that is being claimed here is zeroeth output and the first input is signed using the private key S_2. Hence, the claimer for this previous transaction T_1's output has publickey as P_2 & private key as S_2. Suppose, the claimer is Y

The first output of T_1 is:

```
"out": [
    {
        "value": "10.12",
        "scriptPubKey": "....<hash of P2>..."
    }
]
```

As we know T_1 is valid, so by this output script Y(public key $= P_2$, privatekey $= S_2$) is receiving a value of 10.12

For T_3's second input, we should refer to h_2 which is the hash pointer of transaction T_2 and output index that should refer here is 1st output of T_2. This input is signed using private key S_2. So, it is also being signed by Y. So again Y is the claimer here and they claim that 1st output value of transaction T_2 belongs to them.

P.T.O

First output of transaction T_2:

```
"out": [
    {
        "value": "5.00",
        "scriptPubKey": " ....<hash of P2>...",
    }
]
```

T_2 is a valid transaction, so we need not check its validity. But if we need to check, we can see that someone, say Z, whose public & private keys are P_3 & S_3 respectively, transfers value of 5.00 to Y. Z is receiving some value in transaction T_1's second output, 5.25, and spending 5.00 from its to Y. Z has 0.25 remaining.

So, after seeing T_1 and T_2, we can see that Y has 10.12 from T_1 and 5.00 from T_2. So, Y has 15.12. Y is referring to both this output in input of T_3 and signing it with its private key S_2.

Now in the output of T_3, we can see that,

```
"out": [{
    "value": "15.10",
    "scriptPubKey": "....<hash of P4>...."
}]
```

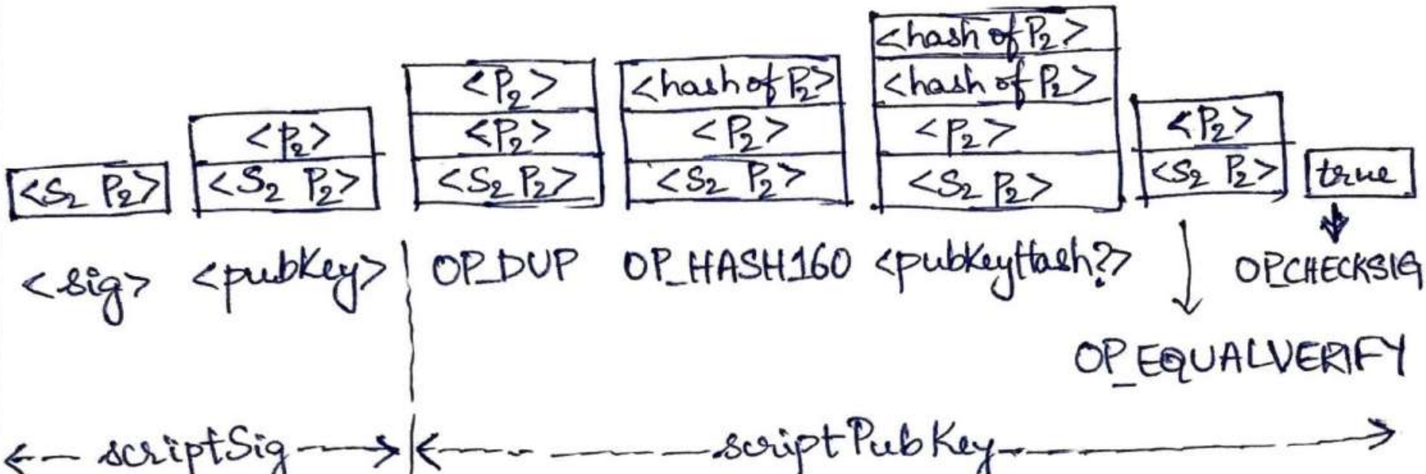
In the output the value 15.10 is sent to someone who has P_4 as its public key that means the value 15.10 is transferred to the address which is hash of P_4. We saw that Y has value 15.12 so Y can send a value of 15.10. This transaction is hence valid. The scripting language does not check validity in this way. It uses stack based execution. To perform that first, a transaction-based ledger is drawn here for simplicity. Suppose a person $\underline{S}$ has a public key P_4 and X has public key P_1 and private key S_1.

T_1	Input: $h_0[0]$ ⤳ (Suppose X had this money already) Output: $10.12 \rightarrow Y$ $5.15 \rightarrow Z$ Signed(X)
T_2	Input: $h_1[1]$ Output: $5.00 \rightarrow Y$ $0.15 \rightarrow Z$ Signed(Z)
T_3	Input: $h_1[0]$, $h_2[0]$ Output: $15.10 \rightarrow S$ $0.02 \rightarrow Y$ Signed(Y)

Transaction-based ledger similar to the one in Bitcoin.

Stack Based Validity Checking of T_3 :

Execution of Bitcoin Script



Here, the first 2 instructions are one data instructions — the signature (<S_{2*} P_2>) and the public key used to verify that signature which is specified in the scriptSig component of the input part of the transaction T_3. As they are data instruction, they are pushed onto the stack.

Now in the scriptPubKey part, that is the output script of T_2, 1st we have OP_DUP which pushes a copy of top element into the stack top. [scriptPubKey of T_1 can also be used to validate]

The next instruction is OP_HASH160 which pops the top value, compute the cryptographic hash and pushes the result onto top of the stack.

Then we will push one more data onto the stack. This data was specified by the sender of the transaction T_2. It is the hash value of a public key that was specified; the corresponding private key must be used to generate the signature to redeem these coins. At this point, there are two values at the top of the stack.

There is the hash of the public key, as specified by the sender and the hash of the public key that was used by the recepient when trying to claim the coins. OP_EQUALVERIFY checks these 2 values at the top. If they are not equal, error occurs, script stops. Else if both of them are equal, the instruction consumes these values. Now, stack has a signature and the public key.

The public key is already checked so we have to check validity of the signature. OP_CHECKSIG pops both the signature & public key and does the signature verification. It then returns TRUE, implying T_3 is a valid transaction as in the same way we can check validity for 1st input as well as output.

Ans 3 The bitcoin transaction T will contain the hash of the block of its inputs and their indices in their respective blocks. So, we could use those block hashes to retrieve the blocks and in turn T's input transactions.

The inputs for T could be like shown here:

```
"in": [ {
          "prev_out": {
          "hash": "h1"        -> block hash of first input of T
           "n" : n1           -> index of first input of T in block h1
       }, "scriptSig": "s1 p1"
    },
    .... // other inputs
  ]
```

We will go to block h_1, see the n_1^{th} transaction.

So, this is how we search for input 1 of T.

Do the same for other inputs as well.

Mechanism to reduce the time & space complexities of search

Time Complexity:

The length of blockhash is always fixed. So, I suggest we use a Trie Data Structure such that the end node will contain a pointer to our block. Now, we can retrieve the value and scriptPubKey to actually validate transaction T.

The time complexity of finding a datablock will take O(n) time [n → length of hash of block]

Space Complexity

Now, there are a large no. of transactions taking place in the Bitcoin blockchain. So, we need to come up with a strategy to reduce the space complexity of the data structure proposed above, for the search operations.

Now, transactions involve 2 accounts: Sender & Receiver. And the new transaction T, would depend on the input transactions which earlier involved one of the accounts.

So, I propose we maintain a Trie for each account. The Trie of an account will only store the block hashes in which a transaction involving that account is present.

This will reduce the space required per account basis to manage.
So, first we will create a Trie DS for each account and traverse the whole blockchain to populate it. So, space complexity is O(m) where m is the length of blockchain.

In case of an update, we will update the Trie of only the involved accounts in O(n) time complexity.

THE END