CS 5433: Blockchains, Cryptocurrencies, and Smart Contracts

Assignment1

Rom Cohen, netID: rc783

Neel Parekh, netID: np423

Robert Wolfe, netID: rjw253

Problem1:

1c:

A problem with the POA we implemented above is that only one actor authority, meaning that if that actor has a different incentive than the rest of the network he could only approve blocks that are good for him.

A way to get passes this is to combine the two implementations we did. We would have a set of actors that will be authorities. For a block to be approved the authorities will need to show proof of work. Assuming that the honest actors have a majority in the set of authorities, we will have a safer protocol than the implemented POA.

Problem 2 - UTXO Management in Wallets

Part 1

Let's assume Alice is a legitimate user of Moonbase. Since Moonbase handles millions of transactions, normally it is likely there will be at least one UTXO that has a value greater than Alice's withdrawal amount. In this scenario, the described protocol works appropriately. In the scenario where a malicious actor with a high balance in Moonbase (let's call her Eve) decides she wants Alice's legitimate withdrawals of money to be denied, Eve can mount a DoS attack by withdrawing money in multiple transactions, each withdrawal at an amount greater than Alice's desired withdrawal amount, until there are no UTXOs of sufficient value available to fund

further transactions. If Eve can complete these withdrawals sufficiently in advance of Alice's withdrawal request, Moonbase's supply of UTXOs that are able fund Alice's withdrawal UTXO will be depleted. Alice will not be able to withdraw any more money until another user deposits more than Alice's desired withdrawal amount (in a single transaction).

In order to prevent DoS attacks in this protocol, Moonbase should fund single withdrawals with an aggregate of input UTXOs. For example, consider the following pseudocode to fund a withdrawal of `withdrawal_amt`:

```
funding_set = { }    # initialize an empty set of UTXOs used to fund the withdrawal
U <--- Randomly choose a UTXO from the pool of UTXOs
funding_set.add( (U, U.value) )

while sum(funding_set.values) < withdrawal_amt:
        U <--- Randomly choose a UTXO from the pool of UTXOs
        funding_set.add( (U, U.value) )

return funding_set
```

Here, we randomly add UTXOs to our input until their aggregate value exceeds the withdrawal amount. By combining UTXOs, as long as Alice's withdrawal request is less than the total holdings of Moonbase (regardless of how the holdings are split up between UTXOs), Moonbase will be able to fund Alice's withdrawal. Further, in order to mount a DoS attack, Eve will now need to own a portion of the Moonbase holdings with value roughly no less than the total aggregate value of Moonbase holdings minus Alice's withdrawal amount. Although not insurmountable, this is significantly more difficult to accomplish.

Part 2:

In order to reduce the number of UTXOs in the database, one strategy is to change the protocol to use the maximum number of UTXOs as possible. We can move in this direction by utilizing the same protocol outlined above, but instead of aggregating a random set of UTXOs until their aggregate value matches or exceeds the withdrawal value, we combine the smallest-valued UTXO in existence with progressively larger-valued UTXOs until the aggregate value of this set of UTXOs matches or exceeds the withdrawal value. This protocol would significantly slow down the system as you would need to do near-continuous sort operations, and there might be certain points where you pointlessly combine two UTXOs (i.e. you want to spend $5 but the only UTXOs are $1 and $15); nevertheless, the protocol maximizes the number of UTXOs used in each transaction, while guaranteeing the same protections as the protocol outlined previously.

Problem 3

1. The protocol satisfies validity when $< \frac{n}{4}$ players are faulty.

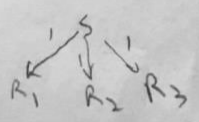
Proof: An honest sender will send the same message m to all recievers

All honest recieves and the sender multicast votes for m to all other players.

Since $< \frac{n}{4}$ players are faulty, all players recieve $\geq \frac{3}{4}n$ votes for m

All honest players output m. ■

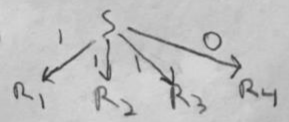
- Example of validity failing for $\frac{n}{4}$ players being faulty:



consider $n=4$. The sender initially sends 1 to the 3 recievers. The one faulty reciever votes 0 in the voting round. The honest recievers then only see 2/4 votes for 1 and output $\perp$. (Note: this is assuming a player doesn't count their own vote in round 3)

2. Let $n=5$. Assume all recievers are honest,

Round 1:



sender sends 1 to R_1, R_2, R_3 and 0 to R_4.

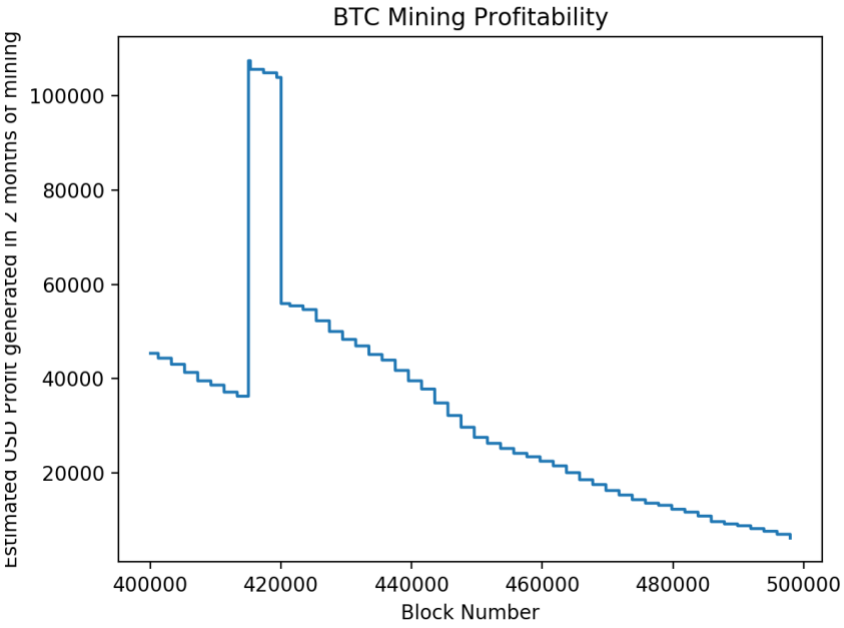
Round 2: If sender sends a vote for 1 to R_4 in Round 2, R_4 will recieve $\frac{4}{5}$ votes for 1. Sender sends votes for 0 to R_1, R_2, R_3.

Round 3: R_1, R_2, R_3 don't recieve $\geq \frac{3n}{4}$ for any output so they output $\perp$

R_4 recieves $\frac{4}{5}n \geq \frac{3}{4}n$ votes for 1 so R_1 outputs 1.

3. If 2 honest players output m_1, m_2 where $m_1 \neq m_2$, that would mean they saw at least $\frac{3}{4}n + \frac{3}{4}n = \frac{3}{2}n$ different votes. However, if $< \frac{n}{2}$ players are faulty, the total number of votes the 2 players could see between them is $< \frac{n}{2} + 2(\frac{n}{2}) = \frac{3n}{2}$, so 2 players couldn't see $\frac{3}{4}n$ votes for different m_1, m_2 s. Therefore, all honest players will either output the same m or $\perp$.

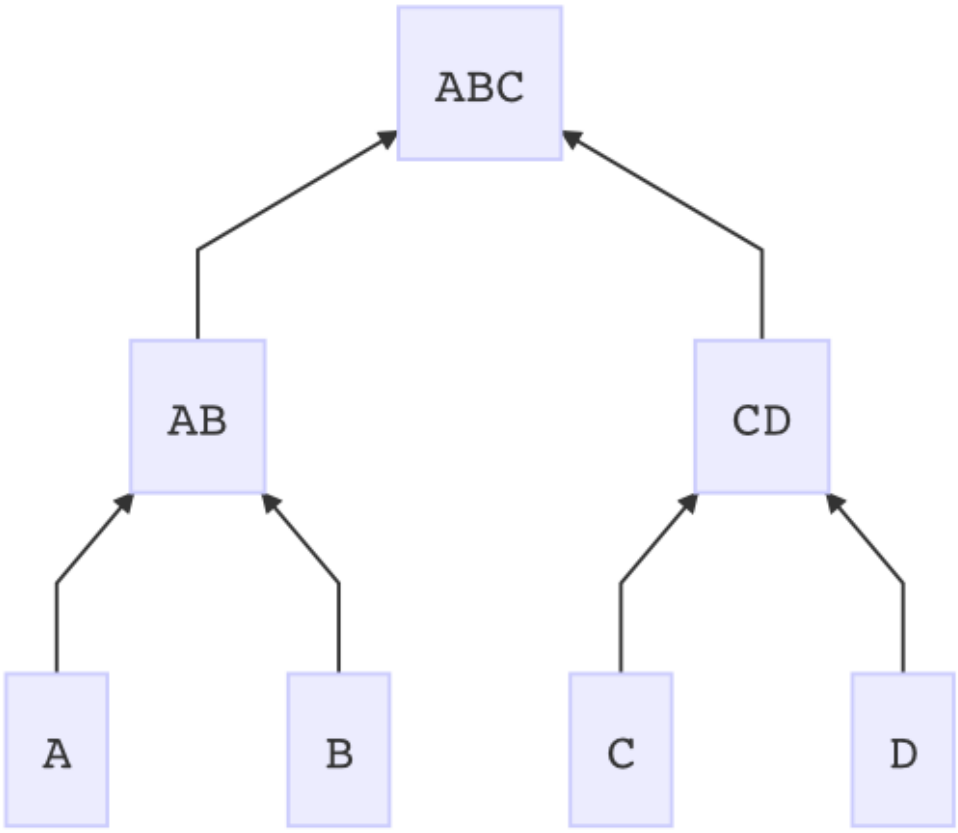
Question 4



Bonus:

a. The solution is implemented in the calculate_merkle_root method in thesubmitted code. The testing code is in tests/testMerkle.py

b. To check inclusion of a transaction for a given merkle root we will need additional information. If we have all of the transactions (except for the given transaction) we could reconstruct the merkle tree, including the given transaction in every possible position in the set. But this is inefficient and gives away a large chunk of data.
To improve this, we can provide the transaction's index, and the hash values of all of the neighboring nodes in the given transaction's "branch" in the tree (a path from the transaction's leaf to the root). With this information we could get to the merkle root of the tree that has the specific given hash values and the given transaction.
For example, let our transactions be T = [A, B, C, D] and we want to check containment of C, we would need to get the values of AB_hash = Hash(AB) and D_hash = Hash(D), and of course the merkle root hash. Then we will verify that given_merkle_root = Hash(AB_hash + Hash(Hash(given_c) +D_hash)).



Evaluation:

- It was appropriately
- We spent about 15 -20 hours
- We feel that there wasn’t enough coding