CS 765: ASSIGNMENT 3

BY

22M0828	22M2110	22M2109
Dhruv Ritesh Jain	Manish Kumar	Anubhav Jana

Part A

Sybil Attack: To mitigate the Sybil attack, the DApp requires users to register with a unique address on the blockchain. Each address is associated with a single user, and multiple identities cannot be created by a single user. Additionally, the DApp can implement mechanisms such as requiring a minimum reputation score or stake to participate in voting, which makes it costly for malicious actors to create multiple identities.

Relevant code in solidity:

```
function addUser(address _address, uint _balance) external {
    require(!users[_address].exists, "User already exists");
    users[_address] = User(_balance, true);
}
```

Evaluation of Voter Trustworthiness:

- The DApp evaluates the trustworthiness of voters based on their voting history and the accuracy of their votes compared to the consensus outcome. It considers factors such as the consistency of votes, the alignment of votes with the consensus, and the reputation of the voter in specific categories of news.
- Relevant parts of the code: The DApp keeps track of each user's voting history and evaluates their trustworthiness based on factors such as their consistency in voting and alignment with the consensus outcome. The updateScoreConfidences function updates the truthfulness scores of voters based on their votes.

```
function updateScoreConfidences(string memory _contentHash) internal {
    // Retrieve news item
    News storage newsItem = newsItems[_contentHash];

    // Calculate weighted score and update news score
    // Determine news status based on the score

    // Define reward factors for voting
    // Update voters' truthfulness scores based on news status
}
```

Weighted Voting:

- The DApp gives more weight to the opinions of trustworthy voters by considering their reputation scores or trust levels. However, it also takes into account the relevance of their expertise to the specific category of news being evaluated.

- Relevant parts of the code: The DApp calculates the total confidence and total rewards for the news, and then distributes the rewards proportionally among the voters. It updates each voter's balance and deducts rewards from the requester's balance based on their votes.

```
function distributeRewards(string memory _contentHash) internal {
    // Retrieve news item
    News storage newsItem = newsItems[_contentHash];

    // Calculate total confidence and total rewards
    // Distribute rewards proportionally
    // Update voters' category trust scores if news status is not Tie
    // Mark news item as finished
}
```

Incentivizing Rational Participation:

- The DApp incentivizes rational participation by rewarding users for voting truthfully and accurately. Users earn rewards based on the accuracy of their votes and their contribution to the consensus outcome.
- Relevant parts of the code: The DApp rewards users for participating in the fact-checking process and voting truthfully. The voteOnNews function updates the user's balance and the distributeRewards function distributes rewards based on their contributions.

```
 function voteOnNews(address _address, string memory _contentHash, int _score) external
{
        // Retrieve news item
        News storage newsItem = newsItems[_contentHash];

        // Update user's balance
        // Distribute rewards based on user's votes
    }

    function distributeRewards(string memory _contentHash) internal {
        // Retrieve news item
        News storage newsItem = newsItems[_contentHash];
```

```
        // Logic to distribute rewards based on user's contributions
    }
```

Identifying News Items:

- News items can be uploaded to the DApp by providing a unique identifier, such as a content hash or a URL, which uniquely identifies the news article or item.
- Relevant parts of the code: The requestFactCheck function allows anyone to request fact-checking for a news article by providing a content hash as a unique identifier.

Bootstrapping Trust:

- Initially, the DApp may not have sufficient data to evaluate the trustworthiness of voters. In such cases, it can bootstrap trust by assigning a neutral or low initial reputation score to all users and gradually adjusting their scores based on their voting behavior and contribution to the consensus outcome.

- Relevant parts of the code: The DApp can start with neutral or low initial reputation scores for all users and adjust their scores over time based on their voting behavior and contributions to the consensus outcome.

```
// Initial reputation scores can be set during user registration or in the addUser function
function addUser(address _address, uint _balance) external {
    require(!users[_address].exists, "User already exists");
    users[_address] = User(_balance, true);
}
```

Part B

Below is the pseudo code of all the functions in Solidity. The internal details of the code is mentioned inside the functions itself as comments.

First, we start by defining the struct of the News.

```
struct News {
        address requester;
        string category;
        string contentHash;
        uint maxVotes;
        uint minimumTrust;
        uint registrationFee;
        uint participationReward;
        uint maxReward;
        uint score;
        bool finish;
        string status;
        bool valid;
        mapping(address => bool) registered;
        address[] voters;
        int[] votes;
        uint[] confidences;
        mapping(address => int) truthfulness;
    }
```

Now we define the events

```
 // Events
    event NewsRegistered(string contentHash, address requester);
    event NewsVoted(string contentHash, address voter, int score);
    event ScoreUpdated(string contentHash, uint newScore);
    event RewardsDistributed(string contentHash);
```

The main functions of the code are as follows:

First is the request fact check function where a user requests the fact checking of a particular news:

```
function requestFactCheck(address _address, string memory _category, string memory _contentHash, uint _maxVotes, uint _minimumTrust, uint _registrationFee, uint _participationReward, uint _maxReward) external {
        require(!newsItems[_contentHash].valid, "News already exists");
        newsItems[_contentHash] = News(_address, _category, _contentHash, _maxVotes, _minimumTrust, _registrationFee, _participationReward, _maxReward, 5, false, "Tie", true);
        emit NewsRegistered(_contentHash, _address);
    }
```

There is also a helper function which provides the functionality to register each user before they start voting for a particular news item.

The code of that function is as follows:

```
 function registerFactCheck(address _address, string memory _contentHash)
external {
        require(users[_address].exists, "User does not exist");
        require(!newsItems[_contentHash].registered[_address], "User already
registered");

require(users[_address].categoryTrustScore[newsItems[_contentHash].category] >=
newsItems[_contentHash].minimumTrust, "User does not meet minimum trust
criteria");
        require(users[_address].balance >= newsItems[_contentHash].registrationFee,
"Insufficient balance to register");

        users[_address].balance -= newsItems[_contentHash].registrationFee;
        newsItems[_contentHash].registered[_address] = true;
    }
```

The actual voting is done using the vote on news function:

```
  function voteOnNews(address _address, string memory _contentHash, int _score)
external {
        require(users[_address].exists, "User does not exist");
        require(newsItems[_contentHash].registered[_address], "User not registered for
this news");
        require(newsItems[_contentHash].valid && !newsItems[_contentHash].finish,
"Voting not allowed");

        newsItems[_contentHash].voters.push(_address);
        newsItems[_contentHash].votes.push(_score);
        uint totalConfidence = users[_address].balance *
users[_address].categoryTrustScore[newsItems[_contentHash].category];
        newsItems[_contentHash].confidences.push(totalConfidence);
        users[_address].balance += newsItems[_contentHash].participationReward;
```

```
        emit NewsVoted(_contentHash, _address, _score);
        updateScoreConfidences(_contentHash);
    }
```

We maintain the confidence score of each voter. This is done to have a weighted opinion of each voter.The function to update the confidence score of each voter is updated as follows:

```
function updateScoreConfidences(string memory _contentHash) internal {
    News storage newsItem = newsItems[_contentHash];

    // Compute the new score
    uint totalScoreWeighted = 0;
    for (uint i = 0; i < newsItem.votes.length; i++) {
        totalScoreWeighted += uint(newsItem.votes[i]) * newsItem.confidences[i];
    }
    newsItem.score = totalScoreWeighted / newsItem.confidences.length;

    // Determine the news status based on the score
    if (newsItem.score > 5) {
        newsItem.status = "Real";
    } else if (newsItem.score < 5) {
        newsItem.status = "Fake";
    } else {
        newsItem.status = "Tie";
    }

    // Define rewards and their reversal only once
    int[11] memory scoreFactors = [-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10];
    int[11] memory reverseScoreFactors = [-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10];

    // Update each voter's category trust score
    for (uint j = 0; j < newsItem.voters.length; j++) {
        address voter = newsItem.voters[j];
        int vote = newsItem.votes[j];
        int reward;

        // Select the correct reward based on the news status
        if (keccak256(abi.encodePacked(newsItem.status)) ==
keccak256(abi.encodePacked("Real"))) {
            reward = scoreFactors[vote];
```

```
        } else if (keccak256(abi.encodePacked(newsItem.status)) ==
keccak256(abi.encodePacked("Fake"))) {
            reward = reverseScoreFactors[vote];
        } else {
            reward = 0;
        }

        // Calculate the new score with clamping to the range [0, 100]
        newsItem.truthfulness[voter] = reward * 5; // assuming update_by = 5
    }
}
```

When the vote on any news item is made and the voting is concluded, we distribute the rewards to all the voters as per their vote. We maintain a scale on which we distribute the rewards. The code to distribute the rewards is as follows:

```
function updateScoreConfidences(string memory _contentHash) internal {
    News storage newsItem = newsItems[_contentHash];

    // Compute the new score
    uint totalScoreWeighted = 0;
    for (uint i = 0; i < newsItem.votes.length; i++) {
        totalScoreWeighted += uint(newsItem.votes[i]) * newsItem.confidences[i];
    }
    newsItem.score = totalScoreWeighted / newsItem.confidences.length;

    // Determine the news status based on the score
    if (newsItem.score > 5) {
        newsItem.status = "Real";
    } else if (newsItem.score < 5) {
        newsItem.status = "Fake";
    } else {
        newsItem.status = "Tie";
    }

    // Define rewards and their reversal only once
    int[11] memory scoreFactors = [-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10];
    int[11] memory reverseScoreFactors = [-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10];

    // Update each voter's category trust score
    for (uint j = 0; j < newsItem.voters.length; j++) {
        address voter = newsItem.voters[j];
        int vote = newsItem.votes[j];
        int reward;
```

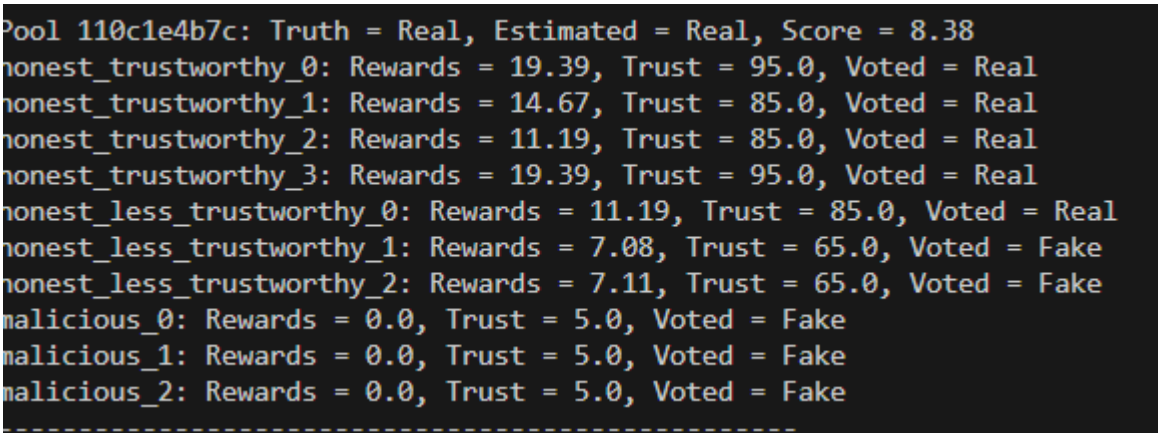
```
        // Select the correct reward based on the news status
        if (keccak256(abi.encodePacked(newsItem.status)) ==
keccak256(abi.encodePacked("Real"))) {
            reward = scoreFactors[vote];
        } else if (keccak256(abi.encodePacked(newsItem.status)) ==
keccak256(abi.encodePacked("Fake"))) {
            reward = reverseScoreFactors[vote];
        } else {
            reward = 0;
        }

        // Calculate the new score with clamping to the range [0, 100]
        newsItem.truthfulness[voter] = reward * 5; // assuming update_by = 5
    }
}
```

This concludes all the functions that we need to implement the fake News detection DApp. Part C contains all the insights that we gathered in various experiments.

Part C

After conducting various experiments, we have made the following two major observations:

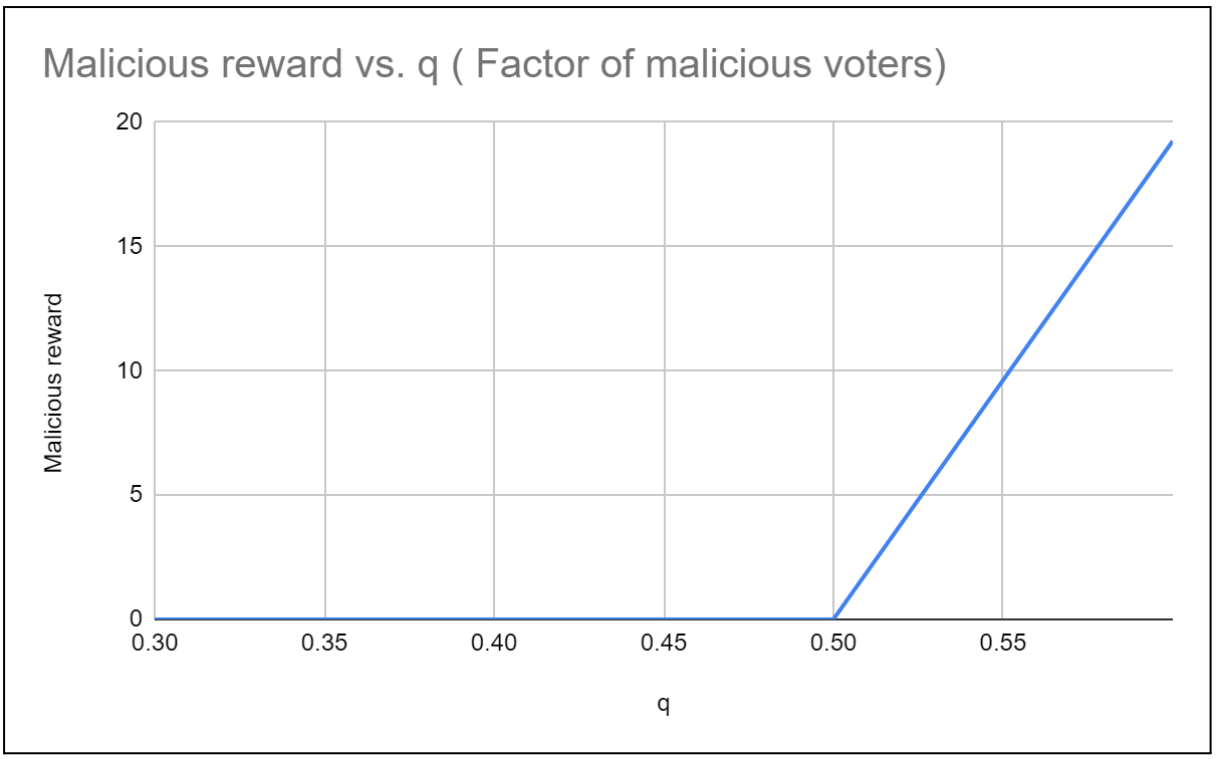
q represents the factor of malicious voters. The simulation depicts that till the time q is less than the majority, the pool consists of honest votes in majority, so the Truth = Real and Estimate is also Real. Also, the malicious rewards are 0.



But as soon as the value of q becomes greater than 0.5, which means that now the majority of voters are malicious, the final pool shows that the Truth = Real and Estimate = Fake.

```
Round 9
Pool 4d9618dbb6: Truth = Real, Estimated = Fake, Score = 0.13
honest_trustworthy_0: Rewards = 0.0, Trust = 0, Voted = Real
honest_trustworthy_1: Rewards = 0.04, Trust = 10.0, Voted = Real
honest_less_trustworthy_0: Rewards = 1.71, Trust = 30.0, Voted = Real
malicious_0: Rewards = 14.04, Trust = 100, Voted = Fake
malicious_1: Rewards = 14.04, Trust = 100, Voted = Fake
malicious_2: Rewards = 14.04, Trust = 100, Voted = Fake
malicious_3: Rewards = 14.04, Trust = 100, Voted = Fake
malicious_4: Rewards = 14.04, Trust = 100, Voted = Fake
malicious_5: Rewards = 14.04, Trust = 100, Voted = Fake
malicious_6: Rewards = 14.04, Trust = 100, Voted = Fake
```

Another thing to note is that the malicious reward is 0 till the majority consists of honest voters, but as soon as the majority becomes malicious, the malicious voters start getting rewards.



The second major observation was the distribution of rewards. Since the total rewards are the same, the reward distribution decreases as the voter population keeps increasing.

Keeping every other factor, i.e. q, p, number of rounds, if we keep increasing the number of voters. Here is the graph for the same.