October 06, 2022

CS61065 Theory and Applications of Blockchain

Assignment 3: Morra Game with Ethereum Smart Contract

Submission deadline: October 14 2022, EOD (Hard Deadline)

You can create a group of two and solve this assignment. Only one member from each group should submit the assignment in Moodle. Clearly mention your group details in the submission.

In this assignment, you will get familiar with Ethereum smart contracts in solidity language. You will learn how you can use blockchain to enforce certain rules in a distributed setting in order to construct applications where the users do not necessarily trust each other.

Submission Instructions
Deploy your smart contract in the Goerli test network. You may use the combination of Remix IDE and MetaMask extension to develop, test, and deploy your contract. **Write the address of the contract and your roll number as comments in the solidity file. Submit your code as a single file with .sol extension as <RollNo>.sol in Moodle.**

Introduction

We want to develop a simple game called ***Morra*****,** which is a hand game that dates back thousands of years to ancient Roman and Greek times. Each player simultaneously reveals their hand, extending any number of fingers, and calls out a number. Any player who successfully guesses the total number of fingers revealed by all players combined scores a point [read more here].

Although the game is straightforward, implementing this over the computer network is not trivial. Imagine you implement Morra by simply passing the number of fingers as a message between the players. Then, between two players P1, and P2, whichever player gives the input first will be at a disadvantage. A similar situation might have occurred in your rock-paper-scissor game where you reveal your choice first, and your opponent then could consciously counter it. Therefore, in a two-player Morra game, somehow the inputs from P1 and P2 must be simultaneous, or in other words, the inputs must be revealed atomically. In this assignment, for simplicity, we will only be considering a **two-party** Morra game.

Morra Implementation Using Public Bulletin Board and Hash Function

An implementation of Morra over the computer network is possible with the help of a *cryptographic hash function*, and a *public bulletin board*. We will be using the cryptographic hash function to design a commitment scheme, where a player can commit its chosen input without revealing it immediately. Then, the public bulletin board will help in enforcing that commitment when the player wants to reveal the actual input.

A cryptographic hash function **h** maps some message **M** into a *hash digest* **H = h(M)**, such that it is practically infeasible to invert or reverse the computation to find **M** from given **H**. It is also practically infeasible to compute another message **M'**, such that the hashes are identical, **h(M') = H**.

We take the help of such a hash function to commit a player's input on the public bulletin board. A player selects some input **I** and a password **P**. Then the commitment message is formed as **h(I||P)**, where || denotes string concatenation. Once the hash is committed, the player cannot change the input to a different **I'**, because it is practically infeasible to find **I'** and **P'** such that **h(I'||P') = h(I||P)**. As a result, the player is bound to reveal the input **I** at the later stage of the game.

When both players have committed their inputs hashes, then they can reveal their inputs in any order, and the outcome of the game can be decided.

During the process of commitment and revealing, the public bulletin board ensures (i) the commitments have consensus among the players and there are some witnesses to back it (ii) non-repudiation of the posted commitments.

Solidity Implementation

In this assignment, you have to implement a two-player Morra game as an Ethereum smart contract with *solidity*. The first player will be throwing its hand, which denotes sending an input between 1 to 5. And the second player will be guessing the value. If the second player is successful in guessing, then it wins, otherwise, the first player wins. Each player makes a bet (some Ethereum) for the game, and the winning player received the total bet amount at the end of the game. The game will have 3 phases:

Initialize

A game has to be initialized by first registering with a bet. Any player can register provided that they're not already registered and that their bet is greater than a fixed minimum of 1 milliether = $\mathbf{10^{-3}}$ **eth**. If the first player has already been registered, a second player wishing to register must place a bet greater than

or equal to the bet of that previous player. This is done to prevent the second player from minimizing its risk by placing a smaller bet. There are no obvious advantages of betting an amount strictly greater than the first player's bet, but one should be free to waste his coins however one wants.

Write a function called `initialize()` as follows:

```
function initialize() public payable returns (uint)
```

- Accepts some value of ethereum.
- Ensure that the input ethereum value is > **10^{-3} eth.** Return 0 otherwise. **Try to use "require" here, and wherever applicable next.**
- Check if some previous initialization was done or not.
- If not previous initialization:
    - Mark the game as initialized.
    - Save the bet amount and address for player 1.
    - Return 1
- If previous initialization exists:
    - Check if bet amount > player 1's bet.
    - Save the bet amount and address for player 2.
    - Return 2
- Ensure that the same player (same account) cannot execute initialization twice.
- On error, return 0

Commit

When the two players have been registered along with their bids successfully, they can start playing by committing their moves. As described previously, the player provides a SHA256 hash of the concatenation of a move, represented by an integer, and a secret password. The contract stores this hash and nobody except the player has access to the actual move input. Once such a hash has been committed, it cannot be modified.

Write a function called `commitmove` as follows:

```
function commitmove(bytes32 hashMove) public  returns (bool)
```

- Ensure that player 1 and player 2 both have registered bids. Return false if this condition is not satisfied.
- The input is SHA256 hash of the concatenation of the move and a password.
    - Example: If move is 3, and password is "abc", then the message will be "3-abc", and SHA256 of the message will be "0x**ece458be434fa064057f04ecfadb9e538ca0edd6b3a4fe3184845d7b617e005a**"
      NOTE: the value of the move can be an integer from 0 to 5.
- If a previous commitment hash already exists by the player, then return false.
- Save the hash of the move of the corresponding player.

- If the executing account is neither player 1, nor player 2, then return false.
- Return true on success, false otherwise.

Reveal

After both player 1 and player 2 have committed their moves, they can reveal their moves. A player reveals what it had played by sending the concatenated move-password string. The smart contract then checks the validity of the revealed move by taking the hash and comparing it with the committed move. If they are equal, the first character of the concatenated string (move, an integer) is saved.

When both the two players have revealed their moves, the two moves are compared. If the moves are equal then player 2 wins. Otherwise player 1 wins. The winning player is paid the total bid amounts. Finally, the variables are reset and the next game can be initialized.

Write a function called `revealmove` as follows:

```
function revealmove(string memory revealedMove) public returns (int)
```

- Validate that the account executing the function is either player 1 or player 2.
- Validate that both players have committed their moves.
- Compute SHA256 hash of `revealedMove` and compare it with the committed move of the player.
- If the hashes match, then save the revealed move of the corresponding player.
- Check if both the players' moves are revealed. If both are revealed, perform the following:
    - Decide the winner player by comparing the move of player 1 and player 2. If the moves are same, then player 2 is the winner, otherwise, player 1 is the winner.
    - Pay the winning player the total bid amount (bid by player 1 + bid by player 2) [this amount should be the same as the balance of the smart contract]
    - Deinitialize - Reset all variables such that the game can be started from the Initialization step again.
- Return the move value if successful, -1 otherwise.

To get the revealed move from the first character of a given string (move-password), you may use the following function:

```
function getFirstChar(string memory str) private pure returns (int) {
    if (bytes(str)[0] == 0x30) {
        return 0;
    } else if (bytes(str)[0] == 0x31) {
        return 1;
    } else if (bytes(str)[0] == 0x32) {
        return 2;
    } else if (bytes(str)[0] == 0x33) {
        return 3;
    }  else if (bytes(str)[0] == 0x34) {
        return 4;
    }  else if (bytes(str)[0] == 0x35) {
        return 5;
    } else {
        return -1;
```

```
        }
    }
```

Functions for debugging:

In addition to the functions for the three phases, also write the following functions to debug and test the smart contract:

`function getBalance() public view returns (uint)`
Return the balance of the smart contract.

`function getPlayerId() public view returns (uint)`
Return 1 for if the executing account is player 1, return 2 for player 2. 0 otherwise.

References

Solidity documentation: https://docs.soliditylang.org/en/v0.8.7/

Remix Editor: https://remix.ethereum.org/