University of Edinburgh

Assignment #4

Student name:

Course: *Blockchains & Distributed Ledgers*
Due date: *January 8, 2023*

Introduction

The below design consists of only one smart contract, and it does not require any additional inheritance of other contracts or libraries. The default rules implemented in the game are the basic rules from wikipedia.

To randomly choose the colours of the players in a secure manner, a commit and reveal scheme is used. The two players first commit their hashes and after reveal the values. Once the second player reveals his value, a random number is computed as a xor of revealed values, and the game is immediately started. The colours of the player's are presented as uint256 1 or 2.

To further improve the game, a timer is added. The timer is involved in all phases of the game. By default, each player has 15 minutes to make a move or pass, and each player also has 15 minutes to reveal.

Throughout the game, players can either make a move or a pass. Every time a player makes a move or a pass, an event is emitted, which notifies the other player.

Values of the intersections on the board are intially set to 0. Once a player places a stone on the board, the value of the intersection changes to either 1 or 2 (depending on the player's colour).

The game can end in two ways. Either the two players consequently pass their turn, or the game is force-quit by any other player once some conditions are fulfilled.

For additional fairness, both players must deposit some amount of ether. After the game ends, both players can withdraw their deposit. There is also mitigation that, if one player decides to quit the game, allows the other to withdraw the deposit and enable a new game to begin.

Variables

Below is the list of the public variables that are used in the design:

- `uint256 stage` is used throughout the game to control which functions can be executed each time. There are seven stages of the game:
  - stages 0 and 1 are the commitment stages of the game
  - stages 2 and 3 are the revealing stages of the game
  - stage 4 is the playing stage of the game. During that time, players are allowed to make a move or pass.

  - stages 5 and 6 are the ending stages of the game.

- `address[] players` is used to declare the two players
- `mapping(address => bytes32) hashes` is used to associate each player with the hash he provides in the commitment stage
- `mapping(address => uint256) deposits` is used to associate each player with the deposit he has sent. Although the deposit for each game is the same, this technique allows the player to withdraw at any time after the game has ended, enabling two other or the same players to start a new game independently of it. It also allows more control over the amount of ether a player can withdraw and further prevents abuses.
- `mapping(address => bool) unlock` is used to control whether the player can withdraw his deposit. It also prevents a player from not revealing after the other player has already revealed.
- `uint256 capturedBlack` and `uint256 capturedWhite` are used to keep track of captured stones
- `uint256 firstValue` is used to store the value that the first player has revealed. It would be optional to declare a mapping that would associate each player with the value he revealed; however, this technique is more gas efficient since the game is immediately started after the second player reveals and therefore does not need his value to be stored at all. It is also efficient for the contract to check whether only the first player has revealed and the second has not, enabling that player to withdraw the deposit.
- `uint8[][] board` is used to present a Go board
- `uint256 turn` is used to keep track of turns. Functions that perform players' moves check if it is the player's turn (e.g., by `require(players[turn%2] == msg.sender)`.It also enables a simple way to check if the pass was made in the turn before (to end the game).
- `uint256 pass` is used to know when the last pass was made. It allows the game's end if one player passes immediately after the other.
- `uint256 timer` is used to prevent one or both players from withholding their turn for too long
- `uint8 capturedOneX, uint8 capturedOneY` and `uint256 capturedOneTurn` are used for the 'ko' rule. A function that checks that a move is not an immediate repetition of the board checks that in the case that in the previous turn, only one stone has been captured and was on this position, it will not result in capturing the stone that captured the one in the previous turn, since it would result in immediate repetition of the board.

Events

- `event Commited (address indexed player)` is used to tell which player has commited
- `event Revealed (address indexed player)` is used to tell which player has revealed
- `event Move (address indexed player, uint8 x, uint8 y)` is used to tell what move a player has made
- `event Winner (address indexed player)` is used to tell which player has won the game

Starting the game

A commit and reveal scheme is used to enable a secure randomized process that decides the colour of two players. A external payable function `commitGame` that takes as a parameter `bytes32 valueHash` does the following:

- check that the player has not already committed a value (double commit)
- check that the `stage` is less than 2
- check that the value of the transaction is one ether. The deposit motivates the players to not abort the game before it ends.
- store the `valueHash` into the mapping `hashes` (`hashes[msg.sender] = valueHash`)
- add the player to the array `players`
- lock the player's deposit (e.g., by `unlock[msg.sender] = false`)
- set the `timer` to 15 minutes from the current time (`timer = block.timestamp + 900`)
- add the value of transaction to the mapping `deposits` (`deposits[msg.sender] += msg.value`)
- increase the `stage by 1`
- emit the event `Commited(msg.sender)`

To reveal the hashes once both players have committed them, an external `revealGame` function that takes parameters `value` and `salt` does the following:

- check that the `stage` is either 2 or 3
- check that the hash of the `uint256 value`, `uint256 salt` and address of the sender is equal to the hash submitted in the commitment stage
- check that the player was not late (by checking `block.timestamp < timer`)

- set the hash of the sender in the mappings `hashes` to initial value (`bytes32(0)`). This prevents a double reveal.
- if `stage` is 2, initialize the variable `firstValue` to `value`. As discussed in the next sections, there is no need to store the second revealed value. This technique is also useful for preventing the second player from not revealing.
- if the `stage` is 3, compute `n` as `n = (value XOR firstValue) % 2` and:
  - if `n` is 0, set the `turn` to 0
  - if `n` is 1, set the `turn` to 1
  - delete `board, capturedWhite` and `capturedBlack`
  - set the `firstValue` to 0
- emit the event `Revealed(msg.sender)`
- increase the `stage` by 1

Playing the game

To make a move, an external function `makeMove` which accepts a `uint8` parameter `x`, and a `uint8` parameter `y` does the following:

- check that the `stage` is 4
- check that the player was not late (by checking `block.timestamp < timer`)
- check that it is a sender's turn (e.g., by checking the `players[turn%2] == msg.sender`)
- check that the move valid (e.g., by `x < 19` and `y < 19`)
- check that the intersection is empty
- check that the move is not a suicide
- if there is a possible repetition of the board ('ko' rule):
  - check that the move is not a ko and peform the capturing. Here the function can e.g., check `if(capturedOneX == x && capturedOneY == y && turn - capturedOneTurn == 1){require(!checkCaptures(x, y))}}`
- else:
  - call the internal `checkCaptures` function to check the captures and remove captured stones
- place the stone on the board (e.g., by `board[move] = uint8(turn%2 +1)`). The inital value of the intersection of the board is 0. Once a player places a stone on the board, the value of the intersection is either 1 or 2, depending on the player's colour.

- increase the `timer` by 900
- increase the `turn` by 1
- emit the event `Move(msg.sender, move)`

To make a pass, an external `makePass` function does the following:

- check that the `stage` is 4
- check that it is a sender's turn (by checking the `players[turn%2] == msg.sender`)
- if the other player has also made a pass in the previous game:
  - increase the `stage` by 1
  - end the game by calling the `endGame` function, which does the area scoring for the player who is calling the function
- if the other player did not pass in the previous turn:
  - initialize `pass` to `turn`
  - increase the `turn` by one
  - increase the `timer` by 900
  - emit the event `Move(msg.sender, 20,20)`

To check the captures, an internal function `checkCaptures` that accepts an `uint8 x, uint8 y` and returns a `bool oneCaptured` does the following:

- check the captures
- set the captured stones' positions to 0
- increase the amount of captured stones in `capturedWhite` or `capturedBlack`
- if only one stone was captured:
  - set the `capturedOneX` and `capturedOneY` to `x` and `y`
  - set the `captureOneTurn` to turn
- return `true` if there has been only one stone captured

Ending the game

To end the game, a public `endGame` function does the following:

- check that `stage` is 5 or 6
- increase the amount of captured stones by area scoring for the player calling the function
- emit the event `Winner(msg.sender)` if the `stage` is 6 (both players have ended the game)

- increase the `stage` by 1
- unlock the deposit so that the sender can withdraw it
- if both players have ended the game set the `stage` to 0 (e.g., by `(if stage = 7){stage = 0}`) and delete array players

To withdraw the deposit, an external `withdraw` function does the following:

- check that either the deposit for the sender is unlocked, either `firstValue` is not 0, and more than 15 minutes have passed since the first player has revealed the value (in case only one player has revealed the value)
- first set a function variable `toWithdraw` to the deposit of the sender (by `uint256 toWithdraw = deposits[msg.sender]`)
- then set the deposit of the sender to 0
- transfer the `toWithdraw` amount of ether to the sender

To prevent the two players from witholding the game forever, an external function `forceQuit` does the following:

- check that either half an hour has passed since the last player has made a move, either there has been more than 411 turns
- optional: unlock the deposits for players of the current game
- set the `stage` to 0
- delete the two players from the array of `players`

Gas efficiency and fairness

- a tradeoff between efficiency and usefulness is the fixed size of the board. The contract would indeed be more useful if it allowed different sizes of boards. However, that would certainly result in a bigger contract size and more gas costs for its deployment and execution.
- another tradeoff between efficiency and usefulness is that the contract allows only one game to be played at a time. Although it would be simple to expand the contract to allow multiple parallel games to be played simultaneously, that would also result in further gas costs for the deployment of the contract and all the operations the players would perform.
- additional tradeoff between efficiency and gas fairness is also that the players need to officially end the game by calling the `endGame` function or making a second pass. A potential solution that lets the players decide on the winner is described in the last section.
- mappings `hashes, deposits`, and `unlock` are used instead of arrays since they are cheaper in gas

- integer variables that are often computed with, such as `capturedWhite` and `capturedBlack` are declared as `uint256` instead of lower uints.
- board is presented in `uint8[][]` double array since its only purpose is to store the location of each player's stones
- addresses in events are declared as `indexed,` as they present lower gas costs than unindexed and are easy to track
- a variable turn has multiple functionalities. It allows one to keep track of which players turn it is, when the pass was made, prevent the immediate repetition of the board (ko rule) and additionally prevent it from going forever (anyone can force quit a game that has more than 411 turns made since the longest game recorded in a survey that involved 50000 games that is what the longest lasted, survey)
- variable `firstValue` is used to store the first revealed value. Although there could be a mapping that associated the players with their revealed values, there is no need to store the second since it is already passed as a parameter into the revel function. The game is immediately started after its execution.
- The second player that reveals the value also pays gas for the computation of the random number. To increase the gas fairness, variables `board, capturedBlack, capturedWhite` and `firstValue` are also deleted by that step. They refund some of the gas to the sender since they release space on the blockchain.
- since the function `makeMove` involves a lot of computation and checks, to make a pass, a separate `makePass` function is declared
- a minor tradeoff is also that the player should not use 0 as a value he uses in the commit and reveal scheme. This is discussed in the next section.

Potential vulnerabilities and mitigations

- as mentioned, the commit and reveal scheme is used as a secure randomized process to decide on players' colours. A potential vulnerability is that one player would not want to reveal his value after seeing the other revealed value. However, that is mitigated by locking the players' deposits until the end of the game. Further mitigation is also implemented if the player who should reveal as the second still does not want to reveal the value, even though he will not be able to withdraw the deposit. In the `withdraw` function, the first player can withdraw the funds if the `firstValue` is not 0 (meaning he did submit the value) and the time for the second player to reveal has already passed. That is done by (`require(unlock[msg.sender] || (firstValue != 0 && block.timestamp > timer)`).
- another potential vulnerability is the reentrancy in the withdraw function. However, that is mitigated by completing all internal work before calling a regular transfer function. The transfer function also prevents the recursive

call of the attacking contract since it possesses a gas limit of 2300 gwei. The mechanism also prevents DoS griefing attacks.

- to prevent a game from getting stuck or going on forever, a forcequit allows anyone to stop the current game if enough time has passed or there have been more than 411 turns made. Optional is that the deposits of the current game players are unlocked.
- to motivate the players into fair play, a deposit of 1 ether is required to join a game. The specific amount is optional and can be decided by the coder.
- to further increase the protection against eclipse attacks, a timer of 15 minutes is set. This duration allows a quick game run; however, passing a longer timer is encouraged to prevent such attacks further.
- timer also prevents block-stuffing attacks. Again it is encouraged to set a longer timer to prevent such attacks.
- front-running and sandwiching attacks are also prevented by the commit and reveal mechanism
- to prevent integer overflow and underflow, a coder should use a 0.8+ version of Solidity or OpenZeppelin's SafeMath library
- since the contract does not depend on its balance, it is immune to a malicious forcible sending ether to it.

Off-chain computation

- Function `checkCaptures` involves the most computation and therefore spends the most gas. To reduce the gas costs, players could perform capturing the stones offline, separate from the chain. An additional function could be `proposeMove`, in which a player could place a stone and submit a list of positions of the stones that he would capture by that move. The other player could then either accept the proposal or deny it. However, if the player denies the proposal, the contract coder would need to decide how the contract should resolve the problem and who would pay for the gas for the execution.
- Another function that involves much computation and, therefore, much gas is the `endGame` function. Similarly, the two players could do the computation locally and then agree on the winner on the chain. One way to implement that could be an external function `agreeOnTheWinner` which would, as an argument, accept the address of the player that the sender proposes as the winner. A function could end the game if both players submitted the same address. However, the coder would need to decide how the contract would resolve the case when the players disagree on the winner.

Appendix

Below is an example of how some of the functions could be implemented.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.16;

contract go {

    uint256 stage;

    address[] players;

    mapping(address => bytes32) hashes;
    uint256 firstValue;

    mapping(address => uint256) deposits;
    mapping(address => bool) unlock;

    uint8[][] board;

    uint256 capturedBlack;
    uint256 capturedWhite;

    uint256 turn;
    uint256 pass;
    uint256 timer;

    uint8 capturedOneX;
    uint8 capturedOneY;
    uint256 capturedOneTurn;

    // black = 0, white = 1

    //================================================================
    // Events
    //================================================================

    event Commited (address indexed player);
    event Revealed (address indexed player);
    event Move (address indexed player, uint8 x, uint8 y);
    event Winner (address indexed player);
```

```
//================================================================
// Functions
//================================================================

 function commitGame(bytes32 valueHash) external payable{
    require(hashes[msg.sender]==0); //double commit
    require(stage < 2);
    require(msg.value == 1 ether);
    stage += 1;
    hashes[msg.sender] = valueHash;
    unlock[msg.sender] = false;
    players.push(msg.sender);
    timer = block.timestamp + 900;
    deposits[msg.sender] += msg.value;
    emit Commited(msg.sender);
}

function revealGame(uint256 _value, uint256 _salt) external {
    require(stage == 2 || stage == 3);
    require(hashes[msg.sender] == keccak256(abi.encodePacked(_value, _salt, msg.sender)));
    require(block.timestamp < timer);
    hashes[msg.sender] = bytes32(0);
    if(stage == 2) {
        firstValue = _value;
    }
    if(stage == 3) {
        uint256 n = (_value ^ firstValue) % 2;
        if(n == 0){
            turn = 0;
        }
        else{
            turn = 1;
        }
        delete board;
        delete capturedBlack;
        delete capturedWhite;
        firstValue = 0;
    }
    stage += 1;
    emit Revealed(msg.sender);

}
```

```
function makeMove(uint8 x, uint8 y) external{
    require(stage == 4);
    require(block.timestamp < timer);
    require(players[turn%2] == msg.sender);
    require(board[x][y]==0);
    require(x < 20 && y < 20);
    if(capturedOneX == x && capturedOneY == y && turn - capturedOneTurn == 1){
        require(!checkCaptures(x, y));
    }
    else(checkCaptures(x,y));
    board[x][y] = uint8(turn%2 +1);
    // require no suicide
    timer += 900;
    turn += 1;
    emit Move(msg.sender, x, y);

}

function makePass() external {
    require(stage == 4);
    require(players[turn%2] == msg.sender);
    if(turn - pass == 1){
        stage +=1;
        endGame();
    }
    else{
        pass = turn;
        turn += 1;
        emit Move(msg.sender, 20, 20);
        timer = block.timestamp + 900;
    }
}

function checkCaptures(uint8 x, uint8 y) internal returns (bool oneCaptured) {
    // check captures
    // set captured stones positions to 0
    // check if only one captured (ko rule)
    // if only one captured{
    // capturedOneX = x, capturedOneY = y;
    // capturedOneTurn = turn;}
}
```

```
    function checkCaptures(uint8 x, uint8 y) internal returns (bool oneCaptured) {
        // check captures
        // set captured stones positions to 0
        // check if only one captured (ko rule)
        // if only one captured{
        // capturedOneX = x, capturedOneY = y;
        // capturedOneTurn = turn;}
    }



    function endGame() public {
        require(stage == 5 || stage == 6);
        // sum all collected stones
        // sum all covered area
        // emit Winner if stage = 6
        stage += 1;
        unlock[msg.sender] = true;
        if (stage == 7) {
            stage = 0;
            delete players;}
    }

    function withdraw() external {
        require(unlock[msg.sender] || (firstValue != 0 && block.timestamp > timer));
        uint256 toWithdraw = deposits[msg.sender];
        deposits[msg.sender] = 0;
        payable(msg.sender).transfer(toWithdraw);
    }

    function forceQuit() external {
        require(block.timestamp - 1800 >= timer || turn > 411);
        unlock[players[0]] = true; // optional
        unlock[players[1]] = true; // optional
        stage = 0;
        delete players;
    }


}
```