BDL Assignment 4

Contract Design

For this implementation of a Go game in Ethereum, a delegation design pattern has been used. There are a total of four contracts and one nested contract.

The contracts are as follows:

1. GoGame
2. GoGameUtility
3. Game
4. Randomness

The Nested Contract is:

Board

Game imports GoGameUtility and Randomness.
Randomness is used in a singleton pattern inside the contract Game.
GoGame inherits Game and Board.
Board is nested within GoGameUtility.

Detailed Description of GoGameUtility Contract

The contract is used as a utility class, which contains several data structures useful in the GoGame contract. It defines the following:

1. An enum *State*, which contains the following enumeration constants:
    a. INACTIVE
    b. IN_LOBBY
    c. READY_TO_REVEAL
    d. REVEALED
    e. LIVE
    f. GAME_OVER

    This enum is used to define the game state in the system.
2. A struct *Player*, which is used to represent a player structure and contains the following private variables:
    a. _address, which is of address type and stores the address of a player
    b. commit, which is of bytes32 type and stores the initial commit used in a commit and reveal scheme for pseudo-randomness
    c. secret, a string which is used to store the secret revealed in the previously mentioned commit and reveal scheme
    d. ready, a bool which stores the readiness of a player to reveal
    e. captured, a uint8 representing stones captured by the player
3. A nested contract *Board* which represents general variables of an arbitrary board state and has the following internal variables:
    a. turn, which is of uint8 type and represents which player's turn it is to make a move
    b. size, which is of uint8 type, and represents the size of the board,
    c. board, which is a 2-dimensional array of uint8 and represents the state of each element on the board

    It has a constructor which initializes the size and board variables. It also has an internal function changeTurn described as follows:

    a. Input(s): None
    b. Outputs(s): None
    c. Suggested Logic: if turn is 1, make turn 2, else make turn 1.

Detailed Description of Randomness Contract

This contract has only one function ***random***, defined as an *external view* type.
Input(s): It takes two strings as input, player1secret and player2secret.
Output(s): A random uint256
Suggested logic: It is suggested that the function concatenates the two secrets and hashes them, and takes the hash based on the current block as well, like the block hash for instance. It then takes a logical OR operation between these two hashes, and hashes it again, and converts the result into a uint256 to return the same.

Detailed Description of Game Contract

Game is a contract that defines the general variables and method needed in any two player game that requires randomness to decide whose turn it is. It imports the GoGameUtility and Randomness contracts.

It defines the following internal variables:

1. player1, of type Player
2. player2, of type Player
3. state, of type State
4. lastCommitBlockNumberWithOffset, of uint256, used to store the offset block number to be checked in the commit and reveal scheme
5. timestamp, of type uint256, used to time moves
6. TIMESTAMP_OFFSET, of type uint8, used to measure the maximum allowed time offset between each move
7. BLOCK_OFFSET, of type uint8 used to calculate the number of offset blocks to be added to the current block number to calculate lastCommitBlockNumberWithOffset
8. DEPOSIT_FEE, of type uint128, representing the amount of fee taken as a deposit to ensure against illegal behaviour in gameplay
9. GAS_FEES of type uint32, collected for gas fees handled by the contract to facilitate gameplay
10. TIMEOUT_FEE, of type uint128, collected to compensate opponent, if player times out and thus forfeits the game
11. commitedHashesMap, a mapping indexing bytes32 type to bool, which keeps track of all the already used commits by players in the commit and reveal scheme

The constructor is used to initialise the values of :

1. TIMESTAMP_OFFSET, with suggested value of 900
2. BLOCK_OFFSET, with a suggested value of 2
3. DEPOSIT_FEE, with a suggested value of 10^{17}
4. GAS_FEES, with a suggested value of 10^{10}
5. TIMEOUT_FEE, with a suggested value of 10^{16}

It has the following events:

1. PlayerDetails, which logs the addresses of player1 and player2
2. PlayerColor, which logs the addresses of the player who got white and the player who got black
3. GameStateChange, which logs the current state of the game, of type State
4. GameOver, which logs the address of the winner
5. Deposit, which logs the address of the sender and the amount received in uint256

It implements the following internal functions:

1. willSatisfyBlockOffset, an internal view function

    a. Input(s): None
    b. Outputs(s): boolean
    c. Suggested Logic: Checks if the current block number is greater than lastCommitBlockNumberWithOffset and returns true if yes. Otherwise, returns false.
2. p1GoesFirst, internal function
    a. Input(s): two strings player1_secret, player2_secret
    b. Outputs(s): boolean
    c. Suggested Logic: Creates an instance of the Randomness class and calls the random function from that instance. Uses a mod 2 on the result received from the function call, and if it is 0, returns true, else returns false. This is a randomised way of deciding which player goes first.

Detailed Description of GoGame Contract

GoGame is the main contract of the system that acts as the engine of the game. It inherits both Game and Board contracts.

It defines the private variables:

1. has_passed, of type bool
2. _timestamp, of type uint256

It has the following events:

1. StoneCaptured, which logs the coordinates of the captured stone on the board and the turn variable value
2. StonePlaced, which logs the coordinates of the captured stone on the board and the turn variable value
3. Passed, which logs the turn variable value
4. RevealOtherPlayer, which is used to signal that the player who has not revealed yet, must reveal in accordance to the block offset rule

It also defines a struct GameData with the following variables:

1. size, of uint8 type
2. board, a 2 dimensional array of uint8 type
3. state, of type State
4. p1_score, of type uint8
5. p2_score, of type uint8
6. turn, of type uint8

It implements the following functions:

1. joinGame, an external payable function
    a. Input(s): commit of bytes32 type
    b. Outputs(s): None
    c. Suggested Logic: Requires that the msg.value is equal to DEPOSIT_FEE + GAS_FEES + TIMEOUT_FEE, and that the game is either in the state IN_LOBBY or INACTIVE, and that the commit has not been used before by checking that commitedHashesMap[commit] returns false. It emits the Deposit event. If the game state is INACTIVE, initialise player1.address with the value msg.sender, initialise player1.commit with the value commit, set the commitedHashesMap[commit] value to true and change game state to IN_LOBBY, and emit event GameStateChange with the state IN_LOBBY. Else, if the state is in IN_LOBBY and the msg.sender is not player1.address, initialise player2.address with the value msg.sender, initialise player2.commit with the

value commit, set the commitedHashesMap[commit] value to true and change game state to READY_TO_REVEAL, and emit event GameStateChange with the state IN_ REVEAL. Also emit the event playerDetails with the variables player1.address and player2.address. Else, revert.

2. freeGame, an external function
    a. Input(s): None
    b. Outputs(s): bool
    c. Suggested Logic: Requires state to be LIVE. If the current block's timestamp is more than the _timestamp + TIMESTAMP_OFFSET, call gameOverSequence function with bool false. Return true.
3. restartGame, an external function
    a. Input(s): None
    b. Outputs(s): bool
    c. Suggested Logic: Require game state to be GAME_OVER. Set the state of the game to INACTIVE, and reinitialise the board to new int8[][](size). Update player1.ready and player2.ready to false. Change the state of the game to INACTIVE and emits a GameStateChange event with the state INACTIVE. Return true.
4. playerReady, an external function
    a. Input(s): message, of type string, stored in memory
    b. Outputs(s): bool
    c. Suggested Logic: Requires game state to be in READY_TO_REVEAL, and
        i. if neither player1 or player2 is ready (refer to ready variable in Player struct), checks which player sent the message, then checks if the hash of the string sent equates to the commit provided earlier, and if satisfied, toggles the ready variable to true. Updates lastCommitBlockNumberWithOffset with the sum of current block number and BLOCK_OFFSET. Emit event RevealOtherPlayer. Return true.
        ii. Else, if at least one of the players is ready, checks if willSatisfyBlockOffset returns true, and then verifies that the sender is player2 (by refering to the address stored), and that the hash of the message provided equates to the commit provided earlier, and if satisfied, toggles the ready variable to true. Updates the lastCommitBlockNumberWithOffset as before. Changes game state to REVEALED. Emits GameStateChanged event with state REVEALED. Return true.
        iii. Otherwise if willSatisfyBlockOffset is false, call gameOverSequence with false bool flag.
5. setBlackAndWhite, an external function
    a. Input(s): None
    b. Outputs(s): None
    c. Suggested Logic: Requires game state to be in REVEALED. Requires current block number to be higher than lastCommitBlockNumberWithOffset. Calls p1GoesFirst function with the secret variable from both the instances of player1 and player2. If the function returns true, set turn to 1. Emit PlayerColor with the addresses of player2 and player1, respectively. Otherwise, set turn to 2, and emit PlayerColor with the addresses of player2 and player1, respectively. The player who does second (The white player) is awarded 6.5 extra points, as per standard komi rules. Then, changes the game state to LIVE, update _timestamp with the timestamp value of the current block and emits GameStateChange with updated game state. Returns true.
6. gameOverSequence, a private function
    a. Input(s): check_Score, a boolean
    b. Outputs(s): None

    c. Suggested Logic: Requires game state to be LIVE. Changes game state to GAME_OVER. If check_Score is false, checks turn, and if turn == 1, transfer DEPOSIT + 2 * TIMEOUT_FEE number of Wei to player2, otherwise, transfer the same amount to player 1. Declare whoever was sent the Wei, the winner. Else, if check_Score is true, call getScore function, check which player has a higher score, and declare him winner. Send both the players an amount equating DEPOSIT + TIME_OUT_FEE. Change game state to GAME_OVER and emit the event GameOver with the address of the winner.

7. pass, an external function
    a. Input(s): None
    b. Outputs(s): bool
    c. Suggested Logic: Require game state to be LIVE, check if it is indeed the turn of the msg.sender by comparing the sender's address with the players to find which player it is, and if it is their turn. If satisfies and if has_passed is true, call gameOverSequence with a false bool. Otherwise, call changeTurn, and change has_passed to true. Eitherway, emit Passed event with the turn variable value. Update _timestamp with the timestamp value of the current block. Return true.
8. getScore, a private view function
    a. Input(s): None
    b. Outputs(s): None
    c. Suggested Logic: Requires game state to be LIVE. Searches the board to see how many unoccupied slots are there around the stones of player1 and player2 (liberties rule), separately. The numbers are then added to the number of stones that each have captured respectively, and the sums returned in an array, player1's sum being at index 0 and player2's being at index 1.
9. getData, an external view function
    a. Input(s): None
    b. Outputs(s): instance of the struct GameData
    c. Suggested Logic: Requires game state to be LIVE. Creates an instance of the struct GameData, collects all the necessary data, and updates the instance, and returns it.
10. makeMove, an external function
    a. Input(s): x-coordiate of type uint8, y-coordinate of type uin8
    b. Outputs(s): bool
    c. Suggested Logic: Requires game state to be LIVE. Checks if it is indeed the turn of the msg.sender by comparing the sender's address with the players to find which player it is, and if it is their turn. If satisfies, then calls placeStone with the x and y coordinates, and returns true. Updates _timestamp with the timestamp value of the current block. Otherwise reverts.
11. placeStone, a private function
    a. Input(s): x-coordiate of type uint8, y-coordinate of type uin8
    b. Outputs(s): None
    c. Suggested Logic: Check if board is empty for the provided coordinates. If satisfies, puts the stone in the coordinate by putting turn (value of the variable turn) into that x, y indices of the board array. Check if after placing, the stone would be surrounded by opponent stones, by calling isSurrounded with the x and y coordinates. If yes, revert, as suicide is not allowed in Go. Emit a StonePlaced event for with the x and y coordinates of the stone, and the value of turn. Call checkCaptures for the x and y coordinates. Finally, irrespective of whether the board was empty or not, call isBoardFull, and if true, call gameOverSequence.

12. isBoardFull, a private view function
    a. Input(s): None
    b. Outputs(s): None
    c. Suggested Logic: Check if board is full. If yes, return true, else false.

13. checkAndCaptureStones, a private function
    a. Input(s): x and y coordinates of the placed stone
    b. Outputs(s): None
    c. Suggested Logic: Check four directions around placed stone (up, down, left, and right). If any adjacent points are occupied by opponent player's stones, check if they are surrounded by calling isSurrounded function with the coordinates of the adjacent point. If surrounded, capture the group of stones by starting at the x and y coordinates of the adjacent point. Emit a StoneCaptured event for every captured stone with the x and y coordinates of the stone, and the value of turn. Record the number of captured stones, and update the board for the indices of the captured stones to 0. Add the number of captured stones to the captured variable for either player1 or player2, depending on whose turn it was.
14. isSurrounded, a private view function
    a. Input(s): x and y coordinates of a stone on the board
    b. Outputs(s): boolean
    c. Suggested Logic: Checks if the group of stones that the stone at the x and y coordinates is a part of, is surrounded by opponent's stones or not. Return true if yes, else return false.
15. forfeitGame, an external function
    a. Input(s): None
    b. Outputs(s): boolean
    c. Suggested Logic: Requires game state to be LIVE. Checks if it is indeed the turn of the msg.sender by comparing the sender's address with the players to find which player it is, and if it is their turn. If satisfies, sets the forfeiting player's captured variable to lowest value possible and calls gameStateOver with true bool. Returns true.
16. receive function, external and payable (receive as alternative to fallback)
    a. Input(s): None
    b. Outputs(s): None
    c. Suggested Logic: Leave empty

Suggestions for Gas Efficiency and Fairness

1. Ensure to do variable packing (variables with lowest space consuming data types declared first in contract)
2. Minimize reading and writing on-chain data
3. Use immutable keyword where applicable (like size of the board, if made static for all games)
4. Use view keyword wherever possible. View type functions when called, result in gas less transactions.
5. Use solidity version 0.8.0+ to avoid declaring separate functions for overflow/underflow checks.
6. Avoid using loops wherever possible, unless in a view function.
7. Process the front-end application off chain. Use web3 libraries for the chosen programming language to detect events, thus maintaining communication with the contract (game state). Also access external function states to ensure access to scores, and other required information.

Suggested attacks to look out for with possible mitigations

1. Denial of Service: This type of attack overloads the servicing capabilities of a contract, thus making it unavailable to the users, and hindering the functionality. To prevent these, the following mitigations are adopted:
    1. High gas cost is a deterrent.
    2. Ensure no unbounded loops are present in code.
    3. To prevent games from getting stuck, block number based security is advised for the commit and reveal scheme, and for a live game, a timing out based system is advised. As a further deterrent, the timed out party is penalised by not refunding them the

deposit and the timing out fee, and the victim is rewarded a complete refund except for gas fee, and is further rewarded a timing out fee from their opponent as compensation of gas fees.

2. Re-entrancy: A re-entrancy attack can happen when a function makes an external call to another untrusted contract. Then, in an attempt to drain tokens or coins, the untrusted contract makes a recursive call back to the original function. These can be mitigated using the Checks-Effects-Interactions pattern as follows:
    1. Checks are performed on the inputs, sender, values and arguments. In this use case, this is implemented in terms of overflow/underflow checks of solidity 0.8.0+, and authority level checking for certain functions which can only be run by the owner of the contract.
    2. Need to enforce effects and update the state accordingly. For this purpose, game state level is always ensured.
    3. Interact with other accounts via external calls or send/transfer.
    4. payable.transfer() is used instead of call().
3. Preimage attack: If sensitive information is stored in a map, an attacker can game the system by figuring our preimages (keys) of information values in that map that are important to them. This is prevented by using the commitedHashesMap implementation discussed in suggested function logic, which prevents reuse of commits or secrets for the commit and reveal scheme.
4. Unexpected Ether: In certain scenarios, ether can be forcibly sent to contracts in unexpected scenarios to game the system. A fallback function is advised to prevent this.
5. Delegate Call exploit: This happens when the attacker figures out a way to change the calling contract's data upon calling the library in a way that benefits them. To mitigate this, try not to use calls of this format: <<library>>.delegatecall().
6. Tx.origin exploit: A transaction origin requirement can be exploited by an attacker by making a transaction reroute a call to a payment function in the original contract when it calls another contract (compromised contract). To prevent this, tx.origin should not used, and only msg.sender should be used to verify source. Bounced calls are not considered.
7. Default visibilities exploit: A contract may have functions that should have been private, but no modifier has been mentioned, leading the function to fall back to the default public visibility. This can allow attackers to call such functions freely and hamper with the system. Make sure to ensure that correct visibility modifiers are assigned to every variable and function.
8. Unchecked Call Return Values: For certain use cases in external calls, the CALL opcode can be used in Solidity. The call() and send() functions return a boolean to indicate if the call was successful. Thus they come with a simple caveat, in that the transaction that executes these functions will not revert if the external call triggered by call() or send() fails.
   The call() or send() will simply return false. When the return value is not checked, the system enters an erroneous state due to the absence of a revert. Make sure to do so, if used at all.
9. Block Timestamp Manipulation: Block timestamps when used as a source of randomness can become a huge vulnerability, when or if the attacker can win the next block(s), thus determining their timestamp, and controlling the randomness into their favour. This is normally mediated by not using block timestamps, or using the 15 seconds rule. In this implementation, block timestamps are only advised to check for timeouts.
10. Default value exploit (e.g., Nomad-Bridge Hack) and control flow disruption: Ensure that default values are initialised properly.
11. Front-running: Frontrunning is an issue where the attacker tries to perform the same action as the victim, but does it sooner, usually, by outbidding the victim's gas. This is prevented by using state control, and turn control of players, and verifying the addresses of player1 and player2 upon each turn made.
12. Overflow/Underflow exploit: Solidity version >=0.8.0 is used, which has an overflow/underflow check, and will revert contract state if any transaction or non-view function call fails.
13. Pseudo-randomness: The pseudo-randomness schemes used in smart contracts can in fact be predictable in certain cases. If the attacker can figure out the pattern in which the pseudo-

randomness is generated, they can exploit the system in their favour. In this use case, the suggested implementation logic of the functions should be a safe enough source of pseudo-randomness.

Suggested Flow of Game as accessed Externally

1. The game should start when joinGame is called by player1 with keccak256 output of their secret, with a message value equalling or increasing the deposit + timeout fee + gas fee.
2. Join room is called by player 2 with keccak256 output of their secret and similar amount
3. Player 1 reveals their secret by calling playerReady() and passing their secret, which is verified by contract by hashing it with keccak256 algorithm and equating with the previously committed value.
4. Player 2 is notified by the revealOtherPlayer event being emitted and listened to. They wait for 2 more blocks to be mined after the current, and reveal their secret by calling playerReady() and passing their secret, which is verified by contract by hashing it with keccak256 algorithm and equating with the previously committed value.
5. Upon waiting for 2 more blocks, setBlackAndWhite is called by anyone of the parties. The game is thus begun.
6. Players play when it is their turn, and they have the option to makeMove, where they provide the x and y coordinates of the slot with respect to the board, where they want to place their stone. They may also choose the pass by calling the function pass. They must do so within 15 minutes after calling setBlackAndWhite, or after every turn, alternately between each player. They are notified by the gameStateChange into Live, and MakeMove or Passed events, when it is their turn. If they do time out, they lose the game, and their deposit, and timeout fees. The opponent gets a full refund (except on gas fees) and is further compensated an amount equal to the timeout fees, from the opponent.
7. If a player wants to resign and it is their turn, they may choose to do so by calling the forfeitGame function. They do not lose their deposit and timeout fee this way.
8. Otherwise, the game gets over when the board gets full, or when both the player pass.

Suggested pseudo-randomness

Use a commit and reveal scheme as described in the suggested logic for functions and the suggested flow of the game.

Suggested timing-out

Use a block-number-based scheme to ensure that the commit and reveal scheme works without exploits, and use a timing out scheme for the game, for each player’s turn, to ensure fairness is maintained.