COMP-4541 – HomeWork 3

Farshid Adavi - 4021305965007

1 Exercise 1

1.1 Variables

– `alice` and `bob`: Ethereum addresses of Alice and Bob.

– `depositAmount`: Amount of Ether each player must deposit.

– `rewardAmount`: Amount of Ether the winner receives.

– `time_interval`: Time interval between steps in blocks.

– `waiting_time`: Waiting time for the second player to register in blocks.

– `aliceDeposit` and `bobDeposit`: Amount of Ether deposited by Alice and Bob.

– `start_game_block_number`: Block number when both players register.

– `gameStart`: Indicates if the game has started.

– `first_register_block_number`: Block number of the first player's registration.

– `gameEnd`: Indicates if the game has ended.

– `isEnoughToPickWinner`: Indicates if there are enough moves to pick a winner.

– `winnerPicked`: Indicates if the winner has been picked.

– `result`: Result of the game (None, Equal, AliceWon, BobWon).

– `moves`: Stores the move made by each player.

– `committed_moves`: Stores the hash of each player's committed move.

– `commit_flag` and `reveal_flag`: Flags to track player actions.

– `moneys`: Stores the amount of money each player can withdraw.

1.2 Functions

1. `change_alices_address_easily_for_testing(address payable _alice)`

   - **Purpose**: This function is for testing purposes only and allows changing Alice's Ethereum address easily.
   - **Parameters**: _alice - The new Ethereum address for Alice.
   - **Usage**: Call this function with the desired new Ethereum address for Alice.
   - **Note**: This function should be removed from the main contract before deployment to the Ethereum blockchain.

2. `change_bobs_address_easily_for_testing(address payable _bob)`

   - **Purpose**: This function is for testing purposes only and allows changing Bob's Ethereum address easily.
   - **Parameters**: _bob - The new Ethereum address for Bob.
   - **Usage**: Call this function with the desired new Ethereum address for Bob.
   - **Note**: This function should be removed from the main contract before deployment to the Ethereum blockchain.

3. `constructor()`

   - **Purpose**: Initializes the contract by setting the initial block numbers for registration and game start.
   - **Usage**: Automatically called upon deployment of the contract to the Ethereum blockchain.
   - **Note**: This constructor function should remain in the contract as it performs necessary initialization tasks.

4. `only_alice_and_bob_can_register() public payable`

   - **Purpose**: Allows Alice and Bob to register for the game by paying 1 Ethereum each.
   - **Usage**: Call this function and send exactly 1 Ether to register for the game.
   - **Conditions**: Registration must occur before the waiting time expires and only Alice and Bob can register.
   - **Effects**: Updates the deposit amount for the registering player and initializes their move and flags.
   - **Checks**: Verifies that the registration amount is exactly 1 Ether.
   - **Game Start**: If both Alice and Bob have registered, the game starts and the start game block number is recorded.
   - **Timeout**: If one player registers and the other does not within the waiting time, the registration resets.

5. `commit(bytes32 h) public payable`

   - **Purpose**: Allows Alice and Bob to commit their move for the game to prevent front-running attacks.
   - **Usage**: Call this function and send a commitment hash `(sha256(move, nonce))` to indicate your move.
   - **Conditions**: Only Alice and Bob can commit. The game must have started, and the commit time interval must not have expired.
   - **Effects**: Sets the commitment hash for the committing player and marks them as committed.
   - **Checks**: Verifies that the player has not already committed and that the commitment time interval has not expired.

6. `reveal(Move move, string memory _nonce) public`

   - **Purpose**: Allows Alice and Bob to reveal their move for the game.
   - **Usage**: Call this function and provide your move and nonce to reveal your commitment.
   - **Conditions**: Only Alice and Bob can reveal. The game must have started, and the reveal time interval must be active. The player must have committed before revealing.
   - **Effects**: Sets the move for the revealing player and marks them as revealed. If both players have revealed, the function calls `pickWinner()` to determine the game outcome.
   - **Checks**: Verifies that the player has not already revealed, the nonce length is correct, and the commitment matches the provided move and nonce.

7. `pickWinner() public`

   - **Purpose**: Determines the winner of the Rock-Paper-Scissors game and allocates the reward accordingly.
   - **Usage**: Call this function to determine the winner of the game after both players have revealed their moves.
   - **Conditions**: The game must have started, and either both players have revealed their moves or the time interval for revealing has expired. Additionally, this function cannot be called again once the winner is picked or the game ends.
   - **Effects**: Sets the winner of the game and allocates the reward accordingly to either Alice, Bob, or evenly if the game is tied. Marks the game as ended and prevents further calls to this function.
   - **Checks**: Checks if both players have revealed their moves, if either player has failed to reveal, or if only one player has registered for the game. Also checks if the function has already been called or if the game has already ended.

8. `withdraw() public`

  - **Purpose**: Allows Alice and Bob to withdraw their respective winnings after the game has ended.
  - **Usage**: Call this function to withdraw the winnings accumulated from the game.
  - **Conditions**: The game must have ended, and the caller must be either Alice or Bob. Additionally, the caller must have winnings available to withdraw.
  - **Effects**: Transfers the winnings to the caller and sets their balance to zero to prevent re-entry. Marks the function as busy to prevent re-entry by other callers.
  - **Checks**: Verifies that the game has ended, the caller is either Alice or Bob, and the caller has winnings available to withdraw.
  - **Security Considerations**: The function prevents malicious attempts to extort winnings or perform re-entry attacks. It also ensures that the winnings are paid out in a single transaction, preventing overflow attacks.

2 Exercise 2

2.1 Variables

- `auctioneer`: Ethereum address of the auctioneer who initiates and manages the auction.
- `commit_time_interval`: Time interval for bidders to register and commit their bids, measured in blocks.
- `reveal_time_interval`: Time interval for bidders to reveal their bids after committing, measured in blocks.
- `start_auction_block_number`: Block number at which the auction begins.
- `number_of_bidders`: Total number of bidders who have registered for the auction.
- `number_of_reveals`: Number of bidders who have revealed their bids.
- `default_price`: Default price set by the auctioneer, initially set to 1 Ether.
- `isEnoughToPickWinner`: Flag indicating if there are enough bids revealed to pick a winner.
- `winnerPicked`: Flag indicating if the winner has been picked.
- `bigest_bid_address`: Address of the bidder with the highest bid.
- `committed_bids`: Mapping of addresses to the hash of their committed bids.
- `bids`: Mapping of addresses to the amount bid by each bidder when revealing.

2.2 Functions

1. `constructor() public`
   17K Gas

2. `set_default_price(uint price) public`
   7K Gas

   - **Purpose**: Allows the auctioneer to set the default price for bidding.
   - **Parameters**: `price` - The new default price in Ether.
   - **Usage**: The auctioneer calls this function to update the default price.
   - **Conditions**: Only the auctioneer can call this function.
   - **Effects**: Updates the default price for bidding.
   - **Checks**: Verifies that the caller is the auctioneer.

3. `registerAndCommit(bytes32 h) public payable`
   First call 49K Gas and then 32K Gas

   - **Purpose**: Allows bidders to register and commit their bids for the auction.
   - **Parameters**: `h` - The hash of the bidder's committed bid.
   - **Usage**: Bidders call this function to register and commit their bids by sending the default price.
   - **Conditions**: The registration must occur before the commit time interval expires, and the bidder must not have already committed a bid.
   - **Effects**: Records the committed bid and increments the number of bidders.
   - **Checks**: Verifies that the registration amount is exactly the default price and the bidder has not already committed a bid.

4. `reveal(uint bid, string memory _nonce) public payable`
   44K Gas

   - **Purpose**: Allows bidders to reveal their bids after the commit phase.
   - **Parameters**: `bid` - The bid amount, _nonce - The nonce used for commitment.
   - **Usage**: Bidders call this function to reveal their bids by providing the bid amount and nonce.
   - **Conditions**: The reveal must occur after the commit phase and before the reveal time interval expires. The bidder must have committed before revealing.
   - **Effects**: Records the bid amount, updates the highest bid if necessary, and checks if enough bids have been revealed to pick a winner.
   - **Checks**: Verifies that the bidder has not already revealed, the nonce length is correct, the bid is non-negative, and the commitment matches the provided bid and nonce.

5. `pickWinner() public`
   19K Gas

   - **Purpose**: Determines the winner of the auction based on the revealed bids.
   - **Usage**: This function is called after all bids have been revealed to determine the winner.
   - **Conditions**: The reveal phase must have ended, and either all bidders have revealed their bids or the reveal time interval has expired.
   - **Effects**: Sets the winner of the auction and marks the winner as picked.
   - **Checks**: Checks if all bids have been revealed and if the function has already been called.

6. `withdraw() public`
   17K Gas

   - **Purpose**: Allows non-winning bidders to withdraw their bids after the auction has ended.
   - **Usage**: Bidders call this function to withdraw their bids after the winner has been picked.
   - **Conditions**: The auction must have ended, and the caller must not be the winner. Additionally, the caller must have a bid available to withdraw.
   - **Effects**: Transfers the bid amount to the caller and prevents re-entry into the function.
   - **Checks**: Verifies that the auction has ended

3 Both Exercise 1 And 2

- Denial of service: No player can force another player to receive money. Each player only receives their own money. Malicious attempts to extort funds within the withdraw function are not possible because each player receives only their allocated funds.

- Re-entrancy: Re-entrancy is prevented since a player's funds become zero before calling fallback functions. Additionally, the withdraw function is locked when called, preventing re-entrancy by the same address. This lock also prevents any disruption that may occur from two calls with different addresses.

- Bribing miner: It's impossible to bribe miners because each player's funds are fixed and cannot be altered.

- Overflow: Attacks through overflow are mitigated because the entire sum of money is withdrawn in a single transaction.

- Front-running attack prevention: There is no overlap between the commit and reveal time intervals, effectively preventing any front-running attacks. This means that no one can manipulate the outcome of the auction by observing and preempting the actions of other participants.

- Non-response prevention: Withdrawal is only possible for bidders who have revealed their bids. This ensures that participants cannot withhold their reveal to obstruct the auction process. Additionally, by setting an appropriate default price, we mitigate the risk of participants not revealing their bids. If the default price is too high, it may deter participation, while if it's too low, it may encourage malicious behavior such as registering multiple addresses and revealing only the lowest bid. By setting the default price at an optimal level, we incentivize participation and discourage non-response tactics. Opting for half of the estimated maximum value as the default price presents a viable option. This approach is strategic because if a participant registers with multiple addresses, once a bid remains unrevealed and the winner's address is disclosed, the total amount expected can still be attained.

- Consumption Gas: In both contracts all functions are finite and there is no loop or recursion in them, which means that there is an upper bound for their gas. In the first contract, all functions can be called a limited number of times, which shows that the entire game process has an upper bound for gas. In the second contract, the upper bound for the consumption gas of a full auction depends on the number of bidders, but we know that every function has an upper bound. In fact, the critical part of this contract was finding the biggest bid, which instead of using a loop, we broke this task in all calls to the reveal function. In this way, the problem of accepting responsibility for the high cost of calling the pickwinner function was also solved.

- To find an upper bound for the consumption gas for each bidder's participation in this auction, we need to consider the gas costs associated with each function call and operation within those functions. Let's break down the gas costs for each step:

  1. **registerAndCommit Function:**
     - Gas cost for executing the function: $\text{Gas}_{\text{registerAndCommit}}$
     - Gas cost for storage (writing to `committed_bids` mapping and incrementing `number_of_bidders`): $\text{Gas}_{\text{storage\_registerAndCommit}}$
  2. **reveal Function:**
     - Gas cost for executing the function: $\text{Gas}_{\text{reveal}}$
     - Gas cost for storage (writing to `bids` mapping and incrementing `number_of_reveals`): $\text{Gas}_{\text{storage\_reveal}}$
  3. **pickWinner Function:**
     - Gas cost for executing the function: $\text{Gas}_{\text{pickWinner}}$
  4. **withdraw Function:**
     - Gas cost for executing the function: $\text{Gas}_{\text{withdraw}}$
     - Gas cost for storage (updating `bids` mapping): $\text{Gas}_{\text{storage\_withdraw}}$
     - Gas cost for external call (transferring Ether): $\text{Gas}_{\text{external\_withdraw}}$

  Adding these gas costs together gives us the total gas cost for each function call. Then, we need to consider how many times each function can be called by each bidder. For example, a bidder can call the `registerAndCommit` function once and the `reveal` function once, assuming they successfully commit and reveal their bid.

Let's denote the upper bound for the gas consumption for each bidder's participation in this auction as $\text{Gas}_{\text{total}}$. It can be calculated as follows:

$$\text{Gas}_{\text{total}} = (\text{Gas}_{\text{registerAndCommit}} + \text{Gas}_{\text{storage\_registerAndCommit}}) + (\text{Gas}_{\text{reveal}} + \text{Gas}_{\text{storage\_reveal}}) + \text{Gas}_{\text{pickWinner}}$$
$$+(\text{Gas}_{\text{withdraw}} + \text{Gas}_{\text{storage\_withdraw}} + \text{Gas}_{\text{external\_withdraw}})$$

We also need to consider the gas costs associated with the constructor function and any other external calls that might occur during the auction process.