The University of Edinburgh

Blockchains & Distributed Ledgers

Assignment #2: Smart Contract Programming Part I (Morra)

Matriculation Number: S1705270

1. High-level Decisions

The smart contract implements a secure Morra game in Solidity, and it is completely ready to be deployed on the Ethereum blockchain. The game consists of the following steps:

I. Two players register and place a bet.
II. Each player chooses a pick, a guess and a password. They send the hash of the concatenated string (i.e. "pick-guess-password") to the smart contract, and the contract will store this value.
III. When both players have committed their moves, they reveal what they have played. To do so, they send their pick, guess and password in clear (e.g. "pick-guess-password"). The contract verifies that the hash of the received input matches the one stored.
IV. When both players have revealed their moves, the contract determines the winner and sends the total bets to the winner. In case of a draw, each player gets their bet back.
V. The game resets and can be played again by new players.

I divided the smart contract into four phases: registration phase, commit phase, reveal phase and lastly result phase.

In registration phase, anyone can register provided that they meet two conditions: they are not already registered and their bet is greater than or equal to the fixed minimum bet, which is set to one finney. If one player has already been registered, the other player must place a bet greater than or equal to the first player's bet. This constraint plays an important role in increasing the fairness of the game because the second player might always try to bet a smaller amount than the first player so that they can minimise risks and maximise gains.

When a player has been successfully registered, they can start playing. The player will write a string that includes their pick, guess and password, which should be in this form: "pick-guess-password". They must use the provided pure function getHash(), which returns a SHA256 hash of a given input, in order to compute the hash value of their string and then enter this hash as an input for play() method. Once the transaction for play() method successfully completes, this hash is now stored in the smart contract. This hash cannot be modified.

The reveal phase can begin if and only if both players have successfully played their moves. A player reveals what number they picked and guessed by sending to the smart contract the original string that consists of all these information (e.g. "pick-guess-password"). The contract gets a SHA256 hash out of this string and checks if this hash matches the stored hash. If they are the same, then it is verified that the player has not changed their pick or guess and that this reveal() transaction is being requested by the correct player (since no one except this player would know the original string including the password). Therefore, the pick, which is the first character of the string, and the guess, which is the third character of the string, are saved. The smart contract keeps waiting until the other player reveals their move so that we can proceed to the next phase: the result phase.

Finally, in the result phase, any one of the two players can execute the method getOutcome() for the contract to find the winner and send the entire betting to the winner. If there is a winner, the winner takes all the rewards, but players get their bet back in case of a draw. Before the rewards get transferred to the player or the players, the smart contract first resets the game, reinitialising all states. The reason is explained in Section 3.

A player who has the access to the blockchain and information stored in it is still not able to have a peek at the other player's move (pick and guess), because the contract only stores the hash of the players' moves but never the original strings. This smart contract can be considered secure in this sense. Also, after the players both commit their moves, they can never alter their moves. If they do, they are not able to successfully complete the reveal() transaction but are directed to exit the game instead. This eliminates all the likelihood of an opponent player sneaking a peek at the other player's move and therefore ensures the fairness of the game. This is based on the assumption that the hash function used is both pre-image resistant and second pre-image resistant. The hash function used in the smart contract, SHA256, has maintained these properties extremely well till today.

2. Gas Evaluation

In a game that I played with one of my fellows, I have found the following.
I am Player 1 and my fellow is Player 2.

- The cost of deploying the smart contract is 1704895 gas for both players.
- The cost of registering as a player is 43091 gas for Player 1 and 62830 gas for Player 2.
- The cost of playing is 45346 gas for Player 1 and 44794 gas for Player 2.
- The cost of revealing is 37443 gas for Player 1 and 37012 gas for Player 2.
- The cost of getting the outcome is 22052 gas for Player 1 and 40990 gas for Player 2.

The cost of deploying the smart contract is the same for both players, and therefore in terms of deploying cost it can be considered completely fair. However, the cost of interacting with the smart contract has a minor difference for different players. However, the cost is very likely to stay between 20K gas and 40K gas, and the two players are unlikely to have a cost difference greater than 20K.

Why is there such difference when both players are actually executing the same operations? First, for functions like play() and reveal(), it is very likely that Player 1 and Player 2 pass different arguments when calling the functions. Unless they choose exactly the same pick, guess and password, they will be passing different parameters to the functions. This causes a difference in gas costs, and since the scenario where the two players have the exactly same strings is extremely unlikely, there is a cost difference for most of the cases. In fact, even though the arguments passed are exactly the same, it is still possible that the cost of executing the functions vary. The reason is that the cost of writing a non-zero value to a storage location (the contract's state variables) that contains a 0 is greater than the cost of writing to a storage location that already contains a non-zero value. Furthermore, the control flow of a function which a player is calling is likely to

run differently depending on the state of the smart contract or depending on some external state such as **this.balance** or **msg.sender**.

Calling a function that does not write to or read from a storage location, or the contract's state variables, is expected to consume exactly the same amount of gas with the same arguments. However, this does not apply to my smart contract of the Morra, because in the contract, all four methods to be executed to play the game write to or read from a storage location.

The most important key in improving the smart contract in terms of cost efficiency is to minimise unnecessary execution in the code. If we improve our code so that we minimise the use of the blockchain as much as possible, we can avoid paying unnecessary gas fees. Therefore, in my smart contract, there are only four methods to be executed and this plays a pivotal role in minimising gas fees for both users, making the contract more cost-effective. In terms of fairness, it is possible to make both players pay exactly the same gas fees, but it requires a series of adjustments, and since the difference is minimal, we can prioritise other areas for improvements.

3. Potential hazards and vulnerabilities & Analysis of the security mechanisms that can mitigate these hazards

1. Integer Arithmetic Errors

A very common hazard in financial software is a mistake in arithmetic operations. In general, smart contracts express numbers as integers due to the lack of floating-point support. Using integers to represent value requires stepping down to small units for sufficient precision, and therefore it gets really easy to get integer arithmetic wrong. Moreover, there are possibilities of integer overflow and integer underflow. For example, subtracting 4 from 3 in an unsigned integer, which is very frequently used in smart contracts, can cause underflow. Also, integer arithmetic can easily lead to a lack of precision when it is not operated really well. So, I believe that it is not a good conventional practice to have many integer computations.

2. Frontrunning

Frontrunnuing is basically overtaking an unconfirmed transaction, which is a result of the blockchain’s transparency property. It is one of the hardest common vulnerabilities to prevent. Before getting included in a block by a miner, all unconfirmed transactions are visible. Some interested parties can simply monitor these transactions and overtake them by paying higher transaction fees. As this practice can be easily automated easily, frontrunning has become quite common in decentralized finance applications.

It is very difficult to protect smart contracts against frontrunning, because in order to do so significant redesign or refactoring is usually required.

3. Reentrancy attack

Reentrancy attack can occur when a program creates a function that makes an external call to another untrusted contract before it finishes all of its internal work. An attacker can very easily exploit this vulnerability because all she needs to do is to get some amount of balance mapped to their smart contract address and create a fallback function that calls **withdraw()**. In order to prevent reentrancy attack, it is a good convention to finish all internal work before calling external functions and using **transfer()** or **send()** instead of **call()**. Following this good practice, my smart contract uses only **transfer()** instead of **call()**, and before paying the winner (executing **transfer()**) it resets the game and re-initialises all states.

4. Missing Parameter or Precondition Checks

One of the most common mistakes that we encounter is to forget to validate the parameters of a method or to not perform essential checks for an operation to be valid. For example, we might not verify that a player has sufficient token balance to execute an operation or that a required input from the player has not been received.

These vulnerabilities can cause serious hazards and be easily exploited by attackers. They are mostly the consequence of an incomplete design process. It is a good conventional practice to have a written specification of all methods, with all the arguments, preconditions and any other requirements.

In case of my contract, the main four functions, register(), play(), reveal() and getOutcome(), always verify if these calls are valid before starting operations. For example, play() checks if the registration phase has successfully been done, and the reveal() function checks if each player's pick and guess are valid. This is a result of writing down all specifications in detail during the design process.

5. Analysis of a fellow student's contract

1. Reentrancy attack

```
function confirm_P1_numbers(uint8 num, uint8 guess, string memory password) public
    nums_in_range(num, guess)
    sender_created_game
    p2_joined_game(msg.sender)
    correct_hash(num, guess, password)
    {
        if (games[msg.sender].p2_num == guess && games[msg.sender].p2_guess != num){
            //player 1 won
            uint8 prize = games[msg.sender].p2_num + num ;
            uint8 deposit_leftover = 5-num;
            msg.sender.transfer((prize + deposit_leftover) * 1000000000000000000);
            games[msg.sender].player2.transfer((10-prize-deposit_leftover) * 1000000000000000000);
        }
        else if(games[msg.sender].p2_num != guess && games[msg.sender].p2_guess == num){
            //player 2 won
            uint8 prize = games[msg.sender].p2_num + num;
            uint8 deposit_leftover = 5-games[msg.sender].p2_num;
            games[msg.sender].player2.transfer((prize + deposit_leftover) * 1000000000000000000);
            msg.sender.transfer((10-prize-deposit_leftover) * 1000000000000000000);
        }
        else {
            //noone won, sending back equal amount of money
            games[msg.sender].player2.transfer(5 * 1000000000000000000);
            msg.sender.transfer(5 * 1000000000000000000);
        }
        reset_game(msg.sender);
    }
```

A snippet of codes from my fellow's smart contract

This is a snippet of codes from my fellow's smart contract that corresponds to my getOutcome() function. This function determines the winner and transfers all amounts of money bet to the winner. I noticed that he calls the reset_game() function after transferring the money. Attackers can exploit the information stored in the state variables while the transfer() method is being called and take all money at the smart contract's address. Attackers might exploit this vulnerability and perform reentrancy attack, which will result in the winner not getting the rewards. This is why in my smart contract, I reset the game, re-initialising all states, before transferring money.

2. Integer Arithmetic Errors

From the snippet of code, we can also see that he performs arithmetic operations using big integers. This can be considered a potential vulnerability as well. There are possibilities of integer overflow and integer underflow. In addition, integer arithmetic can lead to a lack of precision when it is not performed completely and perfectly. So, I believe that it is not a good practice to have many unnecessary integer arithmetic operations, especially with big integers, as shown in my fellow's smart contract codes.

In general, my fellow's smart contract is complete in the sense that the game successfully runs and all transactions are working. However, it has some unnecessary methods that are called as transactions, which is

not great in terms of gas cost efficiency. Also, there is a number of vulnerabilities that can be easily exploited by attackers. In my opinion, with a few improvements, the game could be more securely played.

6. Transaction History

The smart contract of my own Morra game is deployed at the address, ***0x946539751C18D35146C835BA9688A92B7fd2c4A8***.

In this specific round of game, both Player 1 and Player 2 bet 3 ETH.

Player 1 bet 3 ETH and successfully went through registration as a player. The following is the hash of the transaction: ***0x9055b3fa8ddbc8980ea5db51df073cf83df7727944778b4e20b96a5a2c12b9f2***.

Player 1 played 1, guessed 3 and used the password “haha”. Hence, the string that was used to get a hash was “1-3-haha". Player 1 entered this hash value into the field for play() method and successfully completed the transaction. The following is the hash of the transaction: ***0x18a409c2cf3cbd4821ca66b43b89591dcf12435ea1d05daf834d8b95b8022878***.

After Player 2 also completed play() transaction, Player 1 successfully completed reveal() transaction: ***0x52ff989387c6b1088bb0d1b0c0c5803df0e6e725d8a99817d5fa4dbf94cd9c55***.

When both players have revealed their moves, Player 1 was finally able to complete getOutcome() transaction and see the result. Player 1 won because Player 2 indeed played 3 and made a wrong guess. Player 1 won the Morra game, and a total of 6 ETH was transferred to Player 1’s account. Earning 3 ETH through just one game, Player 1 feels great. The following is the hash of the getOutcome() transaction: ***0xee461f538d65f999fa935246debfef8a1bea8dee593b03abe63224dad7f01ec0***.

Figure 1. Transaction of register() by Player 1

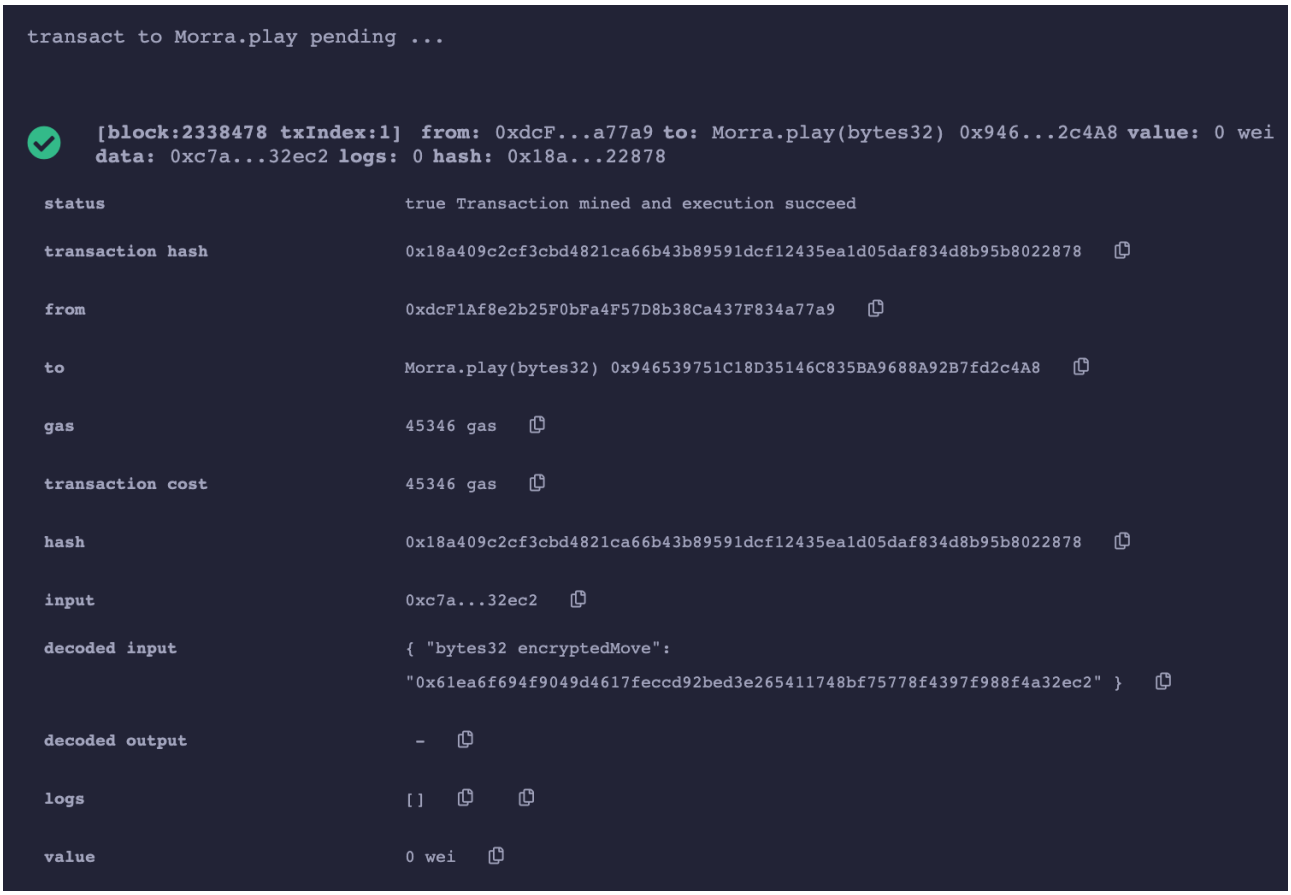

Figure 2. Transaction of play() by Player 1

Figure 3. Transaction of reveal() by Player 1

Figure 4. Transaction of getOutcome() by Player 1

7. Code

```
pragma solidity >=0.4.22 <0.7.0;

contract Morra {

    uint constant public MIN_BET       = 1 finney;    // The minimum bet
    uint public initialBet;                           // Bet of first player

    enum Moves {None, One, Two, Three, Four, Five} // Possible moves (pick or guess)
    enum Outcomes {None, PlayerA, PlayerB, Draw}   // Possible outcomes

    // Players' addresses
    address payable playerA;
    address payable playerB;

    // Encrypted moves
    bytes32 private encryptedMovePlayerA;
    bytes32 private encryptedMovePlayerB;

    // Clear moves; they are set only after both players have committed their encrypted moves
    Moves private pickPlayerA;
    Moves private pickPlayerB;
    Moves private guessPlayerA;
    Moves private guessPlayerB;

    /*
    REGISTRATION PHASE
    */

    // Both players must have not already been registered
    modifier isNotRegistered() {
        require(msg.sender != playerA && msg.sender != playerB);
        _;
    }

    // Bet must be greater than the minimum amount (1 finney)
    // AND greater than or equal to the bet of the first player
    modifier isValidBet() {
        require(msg.value >= MIN_BET);
        require(msg.value >= initialBet);
        _;
    }

    // Register a player.
    // Return player's ID upon successful registration.
    function register() public payable isNotRegistered isValidBet returns (uint) {
        if (playerA == address(0x0)) {
            playerA    = msg.sender;
            initialBet = msg.value;
            return 1;
        } else if (playerB == address(0x0)) {
            playerB = msg.sender;
            return 2;
        }
        return 0;
    }
```

```
/*
COMMIT PHASE
*/

// Player committing must be already registered
modifier isRegistered() {
    require (msg.sender == playerA || msg.sender == playerB);
    _;
}

// Save player's encrypted move (hash).
// Return true if move was valid (there is no encrypted move saved yet), false otherwise.
function play(bytes32 encryptedMove) public isRegistered returns (bool) {
    if (msg.sender == playerA && encryptedMovePlayerA == 0x0) {
        encryptedMovePlayerA = encryptedMove;
    } else if (msg.sender == playerB && encryptedMovePlayerB == 0x0) {
        encryptedMovePlayerB = encryptedMove;
    } else {
        return false;
    }
    return true;
}

// User should use this to get the hash of their string and enter into the input field for the play() method
function getHash(string memory moveToEncrypt) public pure returns (bytes32) {
    bytes32 encrypted = sha256(abi.encodePacked(moveToEncrypt));
    return encrypted;
}

/*
REVEAL PHASE
*/

// Both players' encrypted moves are saved to the contract
modifier commitPhaseEnded() {
    require(encryptedMovePlayerA != 0x0 && encryptedMovePlayerB != 0x0);
    _;
}

// Compare clear move given by the player with saved encrypted move.
// Return the player's pick upon success, exit otherwise.
function reveal(string memory clearMove) public isRegistered commitPhaseEnded returns (Moves) {
    bytes32 encryptedMove = sha256(abi.encodePacked(clearMove));
// Hash of clear input ("pick-guess-password")
    Moves pick            = Moves(getPick(clearMove)); // Actual number the player picked
    Moves guess           = Moves(getGuess(clearMove)); // Actual number the player guessed

    // If the two hashes match, both pick and guess are saved
    if (msg.sender == playerA && encryptedMove == encryptedMovePlayerA) {
        pickPlayerA = pick;
        guessPlayerA = guess;
    } else if (msg.sender == playerB && encryptedMove == encryptedMovePlayerB) {
        pickPlayerB = pick;
        guessPlayerB = guess;
    } else {
        return Moves.None;
    }

    return pick;
}
```

```
// Return player's pick using clear move given by the player
function getPick(string memory str) private pure returns (uint) {
    byte firstByte = bytes(str)[0];
    if (firstByte == 0x31) {
        return 1;
    } else if (firstByte == 0x32) {
        return 2;
    } else if (firstByte == 0x33) {
        return 3;
    } else if (firstByte == 0x34) {
        return 4;
    } else if (firstByte == 0x35) {
        return 5;
    } else {
        return 0;
    }
}

// Return player's guess using clear move given by the player
function getGuess(string memory str) private pure returns (uint) {
    byte thirdByte = bytes(str)[2];
    if (thirdByte == 0x31) {
        return 1;
    } else if (thirdByte == 0x32) {
        return 2;
    } else if (thirdByte == 0x33) {
        return 3;
    } else if (thirdByte == 0x34) {
        return 4;
    } else if (thirdByte == 0x35) {
        return 5;
    } else {
        return 0;
    }
}

/*
RESULT PHASE
*/

// Compute the outcome and pay the winner(s) and return the outcome.
function getOutcome() public returns (Outcomes) {
    if (pickPlayerA == Moves.None || pickPlayerB == Moves.None ||
        guessPlayerA == Moves.None || guessPlayerB == Moves.None) {
            return Outcomes.None;
            // Both players' pick and guess must be valid
    }

    Outcomes outcome;

    if ((pickPlayerA == guessPlayerB && pickPlayerB == guessPlayerA) ||
       (pickPlayerA != guessPlayerB && pickPlayerB != guessPlayerA)) {
        outcome = Outcomes.Draw;
    } else if (pickPlayerB == guessPlayerA) {
        outcome = Outcomes.PlayerA;
    } else if (pickPlayerA == guessPlayerB) {
        outcome = Outcomes.PlayerB;
    }
    address payable addressA = playerA;
    address payable addressB = playerB;
    uint betPlayerA          = initialBet;
    reset();  // Reset game before paying in order to avoid reentrancy attacks
    pay(addressA, addressB, betPlayerA, outcome);

    return outcome;
}
```

```
// Pay the winner(s).
  function pay(address payable addressA, address payable addressB, uint betPlayerA, Outcomes outcome) private {
      if (outcome == Outcomes.PlayerA) {
          addressA.transfer(address(this).balance);
      } else if (outcome == Outcomes.PlayerB) {
          addressB.transfer(address(this).balance);
      } else {
          addressA.transfer(betPlayerA);
          addressB.transfer(address(this).balance);
      }
  }

  // Reset the game.
  function reset() private {
      initialBet      = 0;
      playerA         = address(0x0);
      playerB         = address(0x0);
      encryptedMovePlayerA = 0x0;
      encryptedMovePlayerB = 0x0;
      pickPlayerA     = Moves.None;
      guessPlayerA    = Moves.None;
      pickPlayerB     = Moves.None;
      guessPlayerB    = Moves.None;
  }

   /*
   HELPER FUNCTIONS
   */

  // Return the balance of the contract
  function getContractBalance() public view returns (uint) {
      return address(this).balance;
  }

  // Return player's ID
  function IAm() public view returns (uint) {
      if (msg.sender == playerA) {
          return 1;
      } else if (msg.sender == playerB) {
          return 2;
      } else {
          return 0;
      }
  }

  // Return true if both players have commited a move, false otherwise.
  function bothPlayed() public view returns (bool) {
      return (encryptedMovePlayerA != 0x0 && encryptedMovePlayerB != 0x0);
  }

  // Return true if both players have revealed their move, false otherwise.
  function bothRevealed() public view returns (bool) {
      return (pickPlayerA != Moves.None && pickPlayerB != Moves.None
              && guessPlayerA != Moves.None && guessPlayerB != Moves.None);
  }
}
```