Morra

1. High-level decisions made for contract design

1.1. When and how is the deposit amount of each game decided and committed?

The deposit amount of the game is decided and committed by each of the players upon joining. It is a requirement of joining, that each player sends a value of 1,2,3,4 or 5 ether. This value is equivalent to the number of fingers put in, as per the rules of the game as interpreted. In this implementation it is not possible to put in 0 fingers as this would further amount to a 0 ether joining fee. As this is a condition of joining, it is not possible for any user to join the game without paying a fee, or making a choice of 'fingers' as this is equivalent to the fee paid in units of ether.

Joining the game is the only way a deposit to the contract can be made. A further function 'isJoinable' shows whether the two available spaces in the game are full and determines whether the game is joinable. Therefore, the value sent with the join function is committed, as players cannot then leave, neither can they join twice. The function 'isJoinable' throws an exception stating "You are already playing the game" if you attempt to join, and therefore deposit, twice.

For completeness, the 'makeChoice' function then allows the two players to make their guess of how many fingers the other person has put in, and only gas costs are incurred here. This also cannot be performed twice, as it is a requirement of this function that boolean flag 'hasPlayerMadeChoice' be false (when a choice is made, this is set to true). Therefore this also throws an exception if the user attempts to update their choice (See section 3 on security for more details)

In terms of sending coin back to players in the event of a draw or a win, this amount is decided when the game is played, i.e when the 'disclose' function is called. In the case of a win, this is the remaining contract balance, and in the case of the draw, this is the original amount put in by each player.

1.2. How are winnings sent to the winner?

Once the game has been played, i.e. the disclose function has been called, boolean flags have been assigned in to identify whether the outcome is a win or a draw. It is not necessary to explicitly identify a loss scenario as this is the complement event to that of a win, and is automatically dealt with. Only *after* the game is played, each player is given the access to the correct withdraw function, to obtain the funds that they are entitled to. This has been implemented deliberately as a pull, not a push function to protect against re-entrancy attacks (See Section 3 for security).

If the game results in a **win**, the function 'withdraw_winner' has a modifier 'isWinner' to indicate whether a winner has been identified. If this is true, the 'withdraw_winner' function is then available to be called by the winning address. This function (using modifiers and require statements) will not allow other players, or non-players, to engage with it, throwing up an exception stating either that they are not a player, or that they are not the winner depending on the mode of entry. No further action is taken by the losing player as the withdrawal of winning funds resets the game, and neither withdraw functions are made available to the losing player. The remaining balance of the contract (total amount sent minus gas) is transferred to the winning address when this function is called, and all conditions have been satisfied. Once this is done, respective flags are reset, and the game is reset for both parties. It is not possible to run the withdraw functions twice due to flag changes that are a requirement. There will also be no balance in the contract to withdraw.

1.3. What happens in case of a draw?

If the game results in a **draw**, achieved either by both players guessing correctly, or both players guessing incorrectly, the game will use a 'draw' flag to indicate a draw has occurred. This is a requirement of the 'withdraw_draw' function, thus allowing it to be called. Both players are subsequently given access to the 'withdraw_draw' function and they each have the ability to withdraw their original stake, minus gas costs. Upon calling, the function identifies the player calling it, and transfers the funds to their respective addresses, and all conditions have been satisfied. Once this is done, only the respective players flags are reset, and that

player is no longer a player. Both players have to do this individually, which results in a game reset when both players have withdrawn their funds.

2. Gas Evaluation

2.1. The cost of deploying and interacting with your contract

Gas estimate as performed by Remix IDE:


```
- "Creation": { "codeDepositCost": "1368000", "executionCost": "1453",
  "totalCost": "1369453" }
- "External": { "disclose()": "114650", "getBalance()": "2071", "get_stake()":
  "768", "join()": "42026", "makeChoice(uint256)": "43340", "withdraw_draw()":
  "infinite", "withdraw_winner()": "infinite" }
```


The functions 'withdraw_draw' and 'withdraw_winner' are both indicated here as requiring **infinite gas** in this analysis. One restriction of the estimate made by Remix, is that modifiers are not explicitly taken into account. In this case there are 2-3 modifiers acting on each of these severely restricting input parameters and when employed in the private blockchain, both ran with finite gas (See Section 5). A similar issue was also presented online [1,2]

The gas used in **deploying** the contract was: 1819778 units of gas, gas price = 0 wei.
Transaction ID: 0x918b61564b530dea5be96ada5188500d28a40fd640efee01b9007e679fdc03ef
(Deployed by account: 0xfD229022DFa1fA7be7893D1cf4c45256cb73A8BA)

A breakdown of the costs in a usage of the contract is given in Section 5.

2.2. Contract contribution fairness (re: stakes and outcomes)

In terms of the money that is put into the contract by both players by joining this contract is fair. The contract receives coin from both players, and winning the game means that you will receive this whole pool of money as this is one of the conditions of the game. When each player joins, they each pay the amount of gas required to join.

After that, there is a gas cost associated for each player with making their choices, which is billed individually at the point of calling, so that is also fair. If you choose to check your stake using the 'getStake' function that is down to the player.

The disclose function is not fair in this implementation. The disclose function can be run by either player, but is only required to be run by one player, and that player bears the cost. The function will not run a second time, as the boolean flag 'discloseRun' is required to be false here and once disclose is called, it is reset to true immediately after.

In the case of a draw, the relevant withdraw function is also called by players individually, so this gas cost is given to each player individually as part of their playing of this game. In the case of a win, the winner can withdraw their funds by calling the 'withdraw_winner' function, and so they alone bear the cost of distributing funds and resetting the game. Although they are bearing the costs, it seems more unfair to ask the losing party to contribute to this reset, so in the authors opinion, this was the best path to take here.

2.3. Contract gas cost effectiveness/ fairness

Gas efficiency: The flags that are employed to indicate whether a winner has been identified, or whether a draw has been reached etc. Assignment of variables to values is an expensive process, so I have chosen to use boolean values, that default to false, avoiding initial assignment. In functions, these are set to true only in the correct scenarios, and as such some game scenarios are more expensive than others. For example, a draw is the most expensive way for the game to be played as both players have to run the withdraw function so the total gas used is higher.

By integrating the joining game and choice of number of fingers put in, the need for additional variables has been reduced, in an attempt to be efficient. This does mean that there is some scope for cheating by opponents as the guess values are tied to transactions. An alternative approach to this would be a static stake for the players, and to have a method that determines what refunds they each receive, if any. This requires the use of more variables and more calculations, and therefore would be a more expensive process.

Gas fairness: This game does not *explicitly* split gas costs but is largely fair. The most unfair aspect is that only one player needs to run (and bear the cost of) the disclose function which merely decides the outcome. Who pays this fee is up to the players, but ultimately one person shoulders this cost. One potential improvement would be estimated the gas fees for running the 'disclose' function that plays the game, and got each player to pay this amount forward. However the benefit of this is that each player will have to use withdraw function to receive this back at the end negating the benefit.

Gas usage during a game depends on whether the users use the 'getStake', or 'getBalance' functions. Seeing as both players cannot see who has won until all choices are submitted and the game played, they are incentivised *not* to do this but it does mean that the gas used is dynamic. It would not be fair to charge both players for the optional usage of the other. This could also be a form of vulnerability that could be exploited by the opponent, spending a lot of gas merely to deplete the funds of the other player despite them not receiving the funds as a result.

Finally, gas is paid by the winner to receive the winning balance, but the stakes here are multiples of 1ETH, which is far larger than this gas costs, and it also means that the losing player is not charged again at the point of play. With this implementation, this is the fairest way to play. If each player put in an additional 1ETH from which to subtract the gas costs, and was refunded the remainder at the end, and the prize money kept totally separate, this would further incentivise players to keep usage to a minimum and ultimately be the fairest scheme.

3. Potential hazards and vulnerabilities

Reentrancy

In this game, the vulnerability lies where the game gives money back to the players who wave won or drawn a game. As the game is unaware whether the addresses it is sending funds to is a wallet account or a contract, there is some vulnerability here for reentrancy. The author has attempted to prevent against re-entrancy attacks [3] principally by employing a scheme for funds withdrawal that employs a 'pull over push' method. This means that the contract will give access to withdraw functions for the players, as opposed to just automatically sending funds to their accounts depending on the outcome of the game to help protect against a reentrancy attack [4].

Within the withdraw functions, the author has implemented the Checks-Effects-Interactions pattern [5], where you must first perform checks on your arguments, update the state and then interact afterwards. This is shown in lines 142 and 162 of Appendix A, and in Figure 1. Setting the balance of the players to zero prevents any external calls propagating [6], before allowing players to withdraw manually using the transfer method. This was further chosen above the send or call function, due to concerns over reentrancy with their use [7].

```
function withdraw_winner() external
    isWinner()  // only winner
    isdiscloseRun() // only when play has occurred
{
    require(draw == 0, "Invalid Request");
    balance[winner] = 0;
    winner.transfer(address(this).balance);
    // After winner has withdrawn, reset game parameters
    stakeOfPlayer1 = 0;
    stakeOfPlayer2 = 0;
    player1 = address(0);
    player2 = address(0);
}
```

Figure 1: Extract showings of Check-Effects Interaction patter within the function that awards players their funds in the event of a draw

By use of a transfer method, as opposed to a send or call method. When there are problems, the transfer method raises an exception which will abort the transaction, e.g. if the transfer ran out of gas. The send or call function returns a false value on failing, but does not abort the transaction so this was avoided.

Possible attacks on other players

(i) The 'getBalance' function:

This function, requires that the 'hasPlayerMadeChoice' flags for each player are true, and causes an exception to be thrown if a player tries to do this before the game has been played using the disclose function, which further requires choices to be made before execution. This is very important, as if it is possible for the get balance function to be called once the choices have been made, players may choose to see the balance of the contract allowing them to calculate the amount put in by their counterpart. This would allow them to engineer a win scenario, so this was avoided by the modifiers and boolean flags that are true only when the game has been 'played', This is required for the getBalance function to operate. If the ruction 'disclose' has not been run, the 'getBalance' function will not be available.

(ii) Privacy of choices, and cheating:

To keep player choices private and to prevent front running, one approach would be a commit-reveal scheme, in which players will have their selections hashed in the contract when used in functions and as such, they are still comparable to a value but are not distinguishable to a specific number. There are however, only 5 possibilities, which is a vulnerability to a dictionary style attack. To further remedy this, the players could hash this together with another chosen value (similar to a seed), generated by them to also be included in the hash. This dramatically increases the possibilities and increases the security of the system.

It is also possible for the designer to use a simple hash (using keccak256 or SHA256 or similar) the choices but as detailed below, the author experienced significant difficulty with this and left this in place as a lower level, but functional level of security. Provided here is an extract of code showing how it was intended to be implemented:

```
    modifier isJoinable() {   //compare to list of 1-5
        require(player1 == address(0) || player2 == address(0), "Game is full.");
        require(msg.sender!= player2 && msg.sender != player1, "You are already in the
game");
        require(keccak256(msg.value) == keccak256((1 ether)) ||
                keccak256(msg.value) == keccak256((2 ether)) ||
                keccak256(msg.value) == keccak256((3 ether)) ||
                keccak256(msg.value) == keccak256((4 ether)) ||
                keccak256(msg.value) == keccak256((5 ether)),
                "Your choice of stake (fingers) is not valid, it should be one of 1-5.");
        _;
    }
```

This does not function due to clashing data types in Solidity - this would have required a significant re-write and despite several attempts this was not functional, and a functional game was prioritised.

To further improve security, a commit-reveal scheme could be implemented. In this instance, players would commit their values (choice of fingers) at the start of the game which will be hashed together with their guess of the players other values, and another value chosen at random by the user. Then later once the game has been played, these values can be revealed and used to calculated the funds to be returned to the players where appropriate. This will be a more expensive implementation in terms of gas, and significantly increase the complexity of gameplay - discussed in Section 4, as this was implemented in my partners code.

In this implementation, the author is not aware of a scenario allowing either player to be aware of the address of their opponent, which is a benefit to security. If players are only aware of the address of the contract, even if they are aware that these transactions will appear on the blockchain, they will not know what address to look for and as such, this negates either player looking up the value of the transactions on the blockchain to calculate the value of their opponents guesses.

(iii) Access by non-players

There is a modifier within this code that requires that the address of the person accessing the contract is equivalent to the msg.sender, i.e. the person accessing the code, be one of the two player addresses specified. This was done to protect from outside access, from non-players.

(iv) Funds being held by the contract:
At the end of the contract when both players have winnings in the contract ready to be withdrawn, there are funds waiting to be transferred by each of the players. The first line of protection here is the flags that must be satisfied in order to access these funds.

(v) SafeMath for overflow/underflow
To avoid underflow or overflow errors, the author has throughout the code provided modifiers to functions allowing inputs to be restricted to not only see the game conditions, but also to avoid overflow or underflow errors. It is recognised that the use of SafeMath is a good way to tackle this, particularly where there are arithmetic operations (add, subtract, divide, multiply) that are replaced with counterparts such as 'sub', 'mul' and 'div'[8].

(vi) A best practice improvement:
It is noted that marking functions as untrusted would be good practice where uncertainty exists around security, e.g. "function **untrustedWithdraw**() public {…", but this has not been implemented here [9].

4. A description of your analysis of your fellow students' contracts, including:

4.1. Any vulnerabilities discovered

The contract detailed in Appendix B also details the game of Morra. The logic employed here is different in that each player stakes 5 ETH in the beginning after joining. After the game is played, both players are entitles to some kind of refund depending on the outcome, meaning that both players must run the withdraw function for each game.

The contract employs a commit-reveal scheme which does allow the hashing of players choices, however, this game is complicated to play. Upon entering, the user must make a commit, which is a hash of the player choices. This hash value is then copied and pasted into the join function that allows the user to join and 5 ETH must also be put in the value to be deducted from their account. Then player then has to reveal their values contained with the hash, which verifies their identity. If incorrect values are entered, the game cannot continue. The game can now be played and no further hash functions are used. This game is secure, but complex to use.

The withdraw functions can then be used to pull down your funds from the contract in the event of either a draw or a win. You can find out the address of the winner from the 'winner' function within the code once the game has been played, which may allow the opponent to stall the progress of the function, as the resetting function is not automatic, or tied to the receiving of funds.

4.2. How could a player exploit these vulnerabilities to win the game?

If users can choose to not reset the game, they can further prevent it being played by other people as values are not reset, leaving the contract in a state that is unusable. The game can be halted by one of the players if they do not successfully reveal their values.

In terms of fairness, this game requires that one of the players use the 'play' function, and the gas for this is not split between players fairly. This will not allow a player to win the game when they should not.

When there is a draw in this game, only one of the players needs to call the withdraw_Rebate function, and funds are pushed to both accounts of the players. In this instance, the best practice would be to allow both players to withdraw the correct amount of funds by calling the withdraw manually.

5. The transaction history of an execution of a game of Morra.

Execution of a game of Morra (authors game) with another player on the private ethereum blockchain.

- Player 1 = partner - account address: 0xC7ae278E4d35dcE3dc7b1b6d1E639d6fc81C7759
- Player 2 = author - account address: 0x90559F7Ef6307Dd2295eC9C4F6Fd10D21B8B0E02
- Contract Deployment Address: 0x7Ef5Aa178aCceDD089AE785F70Dea8f2D337b32a
- Conclusion of game: Player 1 won this game, and was awarded the whole value of the contract.

Description	Transaction ID	Cost details	Address Details
Player 1 joins	0xde9f1855e97d762d3457f47ba5d68892ef01a20fb51ed22de0f5b99a1831bf6b	62999 @ 0 wei gas price Amount Sent 1: ether	From: 0xC7ae278E4d35dcE3dc7b1b6d1E639d6fc81C7759 To: 0x7Ef5Aa178aCceDD089AE785F70Dea8f2D337b32a
Player 1 makes their choice	0xe90d233eb8ad7e4d5e661582c8953cfdc573eade7163e5b0745a3fb8016c5ce1	63673 gas units @ 0 wei gas price Total cost 0 ether	From: 0xC7ae278E4d35dcE3dc7b1b6d1E639d6fc81C7759 To: 0x7Ef5Aa178aCceDD089AE785F70Dea8f2D337b32a
Player 2 joins	0x1e11e8f1b2f6cc598b8d895380f2f62d826ee8e9dfe0acf0d4705aac42b686d4	63258 gas units @ 0 wei gas price Amount sent: 2 ether	From: 0x90559F7Ef6307Dd2295eC9C4F6Fd10D21B8B0E02 To: 0x7Ef5Aa178aCceDD089AE785F70Dea8f2D337b32a
Player 2 makes their choice	0x803658622cb4c8ea186edbfce585d14601dd09dfb0d380a026c47466623edb3c	49488 gas units @ 0 wei gas price Total cost 0 ether	From: 0x90559F7Ef6307Dd2295eC9C4F6Fd10D21B8B0E02 To: 0x7Ef5Aa178aCceDD089AE785F70Dea8f2D337b32a
Player 2 discloses results	41059 0xd76ca6d41466c0507e457d6b143f7e483df4888dafa113c5dff45425ec557583	41059 gas units @ 0 wei gas price Total cost 0 ether	From: 0xC7ae278E4d35dcE3dc7b1b6d1E639d6fc81C7759 To: 0x7Ef5Aa178aCceDD089AE785F70Dea8f2D337b32a
Player 1 collects winnings (game was a win/lose this occasion)	0x53136ae080c007175dd1dc08de4d0fc4587294a6ea4a86b931145060692d67ba	28147 gas units @ 0 wei gas price Total cost 0 ether Amount Received: 3 ETH	From: 0xC7ae278E4d35dcE3dc7b1b6d1E639d6fc81C7759 To: 0x7Ef5Aa178aCceDD089AE785F70Dea8f2D337b32a

6. The code of authors contract.

6.1.Instructions for play.

(i) Joining the game: Enter an ether value into the value field, which must be 1,2,3,4, or 5. Call function 'join'. The function getStake will confirm to you the amount you have submitted (in wei, 1ETH=1e18 wei), which is also the number of fingers you have chosen to put into the game.

(ii) Making your guess: This is your guess of the value entered by your opponent. This must also be 1,2,3,4, or 5. Enter this value into the makeChoice function, and call it. Once these steps are complete you must wait for your opponent to also complete these steps.

(iii) Get results: either player may call the 'disclose' function - this will reveal the results of the game and provide you with an option to withdraw your funds. (a) Should the game be a draw, or neither player guesses correctly, you are able to withdraw your original funds using 'withdraw_draw'. (b) Should you win, the game will allow you to withdraw your winning funds by simply calling the 'withdraw_winner' function. This function will place the remaining balance of the contract into your account.

6.2. Authors code given in Appendix A

6.3. Partners code given in Appendix B

7. References

[1] https://github.com/ethereum/solidity/issues/3779,

[2] https://ethereum.stackexchange.com/questions/62989/remix-ide-infinite-gas-problem-with-withdraw-method

[3] https://quantstamp.com/blog/what-is-a-re-entrancy-attack

[4]https://solidity.readthedocs.io/en/v0.6.11/common-patterns.html#withdrawal-pattern

[5]https://solidity.readthedocs.io/en/v0.5.1/security-considerations.html#re-entrancy

[6]https://solidity.readthedocs.io/en/v0.6.11/security-considerations.html

[7] https://solidity.readthedocs.io/en/v0.6.11/security-considerations.html

[8]https://ethereumdev.io/using-safe-math-library-to-prevent-from-overflows/

[9] https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21#:~:text=A%20reentrancy%20attack%20can%20occur,before%20it%20resolves%20any%20effects.&text=This%20simplest%20example%20is%20when,and%20exposes%20a%20withdraw%20function

Appendix A - Authors Solidity code and Deployment Address

Authors Code:

Deployment address: 0x7Ef5Aa178aCceDD089AE785F70Dea8f2D337b32a

Authors Solidity code: morra.sol

```
1.  pragma solidity ^0.5.0;
2.
3.  contract Morra {
4.      address payable private player1;
5.      address payable private player2;
6.      address payable private winner;
7.      uint private choiceOfPlayer1; //number of fingers guessed player 1
8.      uint private choiceOfPlayer2; // number of fingers guessed player 2
9.      uint private draw; // integer flag indicating presence of a draw
10.     bool private discloseRun;
11.     bool private hasPlayer1MadeChoice;
12.     bool private hasPlayer2MadeChoice;
13.
14.     // When a player joins the game, they have to pay a playing fee equal to their number
    of fingers
15.     uint private stakeOfPlayer1;
16.     uint private stakeOfPlayer2;
17.
18.     mapping(address => uint256) balance;
19.
20.     // The constructor initialise the environment
21.     constructor() public {
22.         assert(1 ether == 1e18);  //specify units for the whole contract
23.     }
24.
25.     // Modifiers
26.     modifier isPlayer() {
27.         require(msg.sender == player1 ||
     msg.sender == player2, "You are not playing this game.");
28.         _;
29.     }
30.
31.     modifier isJoinable() {   //compare to list of 1-5
32.         require(player1 == address(0) || player2 == address(0), "Game is full.");
33.         require(msg.sender != player2 && msg.sender !
    = player1, "You are already in the game");
34.         require(msg.value == (1 ether) ||
35.                 msg.value == (2 ether) ||
36.                 msg.value == (3 ether) ||
37.                 msg.value == (4 ether) ||
38.                 msg.value == (5 ether),
39.                 "Your choice of stake (fingers) is not valid, it should be one of 1-5.");
40.         _;
41.     }
```

```
42.
43.     modifier isValidChoice(uint _playerChoice) {
44.         require(
45.                 _playerChoice == 1 ||
46.                 _playerChoice == 2 ||
47.                 _playerChoice == 3 ||
48.                 _playerChoice == 4 ||
49.                 _playerChoice == 5 ,
50.                 "Your choice is not valid, it should be one of 1-5.");
51.         _;
52.     }
53.
54.     // Checking if our players have made a choice before we can call disclose...
55.     modifier playersMadeChoice() {
56.         require(hasPlayer1MadeChoice && hasPlayer2MadeChoice, "The player(s) have not made
    their choice yet.");
57.         _;
58.     }
59.
60.     modifier isDraw(){
61.         require(draw == 1 && winner == address(0), "There is a draw -
    refunds available");
62.         _;
63.     }
64.
65.     modifier isWinner() {
66.         require(msg.sender == winner, "You arent the winner");
67.         _;
68.     }
69.
70.     modifier restrictBalance(){ //
   modifier added to attempt to eliminate infinite gas costs of withdraw functions
71.         require(address(this).balance >=10 ether);
72.         _;
73.     }
74.
75.
76.     modifier isdiscloseRun(){
77.         require(discloseRun == true, "Not authorised: Play has not finished");
78.         _;
79.     }
80.
81.     // Functions
82.
83.     function join() external payable
84.         isJoinable() // To join the game, there must be a free space
85.     {
86.         if (player1 == address(0)){
87.             player1 = msg.sender;
88.             stakeOfPlayer1 = msg.value; //*1000000000000000000
89.
90.     } else {
```

```
91.             player2 = msg.sender;
92.             stakeOfPlayer2 = msg.value; //*1000000000000000000
93.             }
94.     }
95.
96.     function get_stake() public view returns(uint) {
97.         if (msg.sender == player1){
98.             return stakeOfPlayer1;
99.         } else {
100.            return stakeOfPlayer2;
101.        }
102.    }
103.
104.    //
105.    function getBalance() public view returns(uint256)
106.        {
107.            require(msg.sender == player1 || msg.sender == player2, "Not authorised.");
108.            require(hasPlayer1MadeChoice && hasPlayer2MadeChoice, "Players Havent made cho
    ices yet");
109.            require(discloseRun ==  true, "Balance cannot be viewed before play");
110.
111.            return (address(this).balance);
112.        }
113.
114.    //
115.    function makeChoice(uint _playerChoice) external
116.        isPlayer()                      // Only the players can make the choice
117.        isValidChoice(_playerChoice)    // The choices should be valid
118.    {
119.
120.        if (msg.sender == player1) {
121.            require(hasPlayer1MadeChoice == false, "You have already made a choice");
122.        } else if (msg.sender == player2) {
123.            require(hasPlayer2MadeChoice == false, "You have already made a choice");
124.        }
125.
126.        if (msg.sender == player1 && !hasPlayer1MadeChoice) {
127.            choiceOfPlayer1 = _playerChoice * 1000000000000000000;
128.            hasPlayer1MadeChoice = true;
129.        } else if (msg.sender == player2 && !hasPlayer2MadeChoice) {
130.            choiceOfPlayer2 = _playerChoice * 1000000000000000000;
131.            hasPlayer2MadeChoice = true;
132.        }
133.    }
134.
135.
136.    function withdraw_winner() external
137.        isWinner()  // only winner
138.        isdiscloseRun() // only when play has occurred
139.
140.    {
141.        require(draw == 0, "Invalid Request");
```

```
142.         balance[winner] = 0;
143.         winner.transfer(address(this).balance);
144.
145.         // After winner has withdrawn, reset game parameters
146.         stakeOfPlayer1 = 0;
147.         stakeOfPlayer2 = 0;
148.         player1 = address(0);
149.         player2 = address(0);
150.     }
151.
152.
153.     function withdraw_draw() external
154.         isPlayer()  // only players can access
155.         isDraw() // there must be a draw
156.         isdiscloseRun() // only when play has happened
157.     {
158.         // Reset parameters of the game as players withdraw their entitlements
159.
160.         if (msg.sender == player1){
161.             balance[player1] = 0;
162.             player1.transfer(stakeOfPlayer1);
163.             player1 = address(0);
164.             stakeOfPlayer1 = 0;
165.         } else if (msg.sender == player2) {
166.             balance[player1] = 0;
167.             player2.transfer(stakeOfPlayer2);
168.             player2 = address(0);
169.             stakeOfPlayer2 = 0;
170.         }
171.     }
172.
173.
174.     function disclose() external
175.         isPlayer()          // Only players can disclose the game result
176.         playersMadeChoice() // Can only call disclose (results) AFTER choices are made
177.
178.     {
179.         // Disclose the game result
180.         require(discloseRun == false);
181.         discloseRun = true;
182.         if ((choiceOfPlayer2 == stakeOfPlayer1 && choiceOfPlayer1 == stakeOfPlayer2) ||
183.               (choiceOfPlayer2 != stakeOfPlayer1 && choiceOfPlayer1 !
    = stakeOfPlayer2)) {
184.             draw = 1; // flag to indicate draw
185.             winner = address(0); // no winner
186.
187.         } else if (choiceOfPlayer2 !
    = stakeOfPlayer1 && choiceOfPlayer1 == stakeOfPlayer2) {
188.             winner = player1;
189.
190.         } else if (choiceOfPlayer2 == stakeOfPlayer1 && choiceOfPlayer1 !
    = stakeOfPlayer2) {
```

```
191.             winner = player2;
192.         }
193.
194.         // Reset the guesses - other parts reset as players withdraw
195.         choiceOfPlayer1 = 0;
196.         choiceOfPlayer2 = 0;
197.         hasPlayer1MadeChoice = false;
198.         hasPlayer2MadeChoice = false;
199.     }
200. }
```

Appendix B - Code and Deployment Address of Partner

Partners Deployment Address as used:
0xd57D428EB25937D50E30bF58f03B9039791bBF17

Partners code:

```
pragma solidity 0.5.1;

contract Project{
    using SafeMath for uint256;

    mapping(address => uint256) balance; //to store the balance of each player
    mapping(address => bytes32) private commitments; // to store commitments
    mapping(uint256 => uint256) private states;  // Possible prizes for each sum
    mapping(uint256 => valuesP1P2) Fvalues; //to store the values given of each player

    address payable private player1;
    address payable private player2;

    address public winner;
    address private loser;
    uint256 private stake = 5 ether; // to join the game you have to send the contract 5 e
ther
    uint256 private prize;
    uint256 private result;
    uint256 private value1;
    uint256 private value2;
    uint256 private valueguess;
    uint256 private valueguess2;

    bool private gameOver = true;
    bool private can_reset;
    bool private revealP1 ;
    bool private revealP2;
    bool private losercan;
    bool private canReveal;

    struct valuesP1P2{
        uint256 value;
        uint256 guessvalue;
    }

    constructor () public {
        states[2] = 7;
        states[3] = 8;
        states[4] = 9;
        states[5] = 10;
        states[6] = 10;
        states[7] = 10;
        states[8] = 10;
        states[9] = 10;
```

```
45.         states[10] = 10;
46.
47.     }
48.
49.     function CreateCommit(uint256 value, uint256 guessvalue) external pure returns (bytes3
    2) {
50.             require(value >=1 && value <=5, "Use a value between 1 and 5");
51.             require(guessvalue >=1 && guessvalue <=5, "Use a guess value between 1 and 5")
    ;
52.             return keccak256(abi.encodePacked(value, guessvalue));
53.     }
54.
55.
56.     function Join_game (bytes32 commitment) payable public {
57.         require(player1 == address(0) ||
     player2 == address(0),"Both players have been selected");
58.         require(msg.value == stake, "Pay 5 Ether to join");
59.         require(player1 != msg.sender, "Change address");
60.
61.         if (player1 == address(0)){
62.             player1 = msg.sender; //fill the first player
63.             commitments[player1] = commitment;
64.         }else{
65.             player2 = msg.sender; // fill the second player
66.             commitments[player2] = commitment;
67.             canReveal = true;
68.         }
69.
70.     }
71.
72.     function reveal (uint256 value, uint256 guessvalue) public {
73.         require(canReveal == true, "Both players must commit");
74.         require (msg.sender == player1 || msg.sender == player2, "Invalid player");
75.         require(!revealP1 || !revealP2 , "Choices revealed");
76.
77.         if(msg.sender == player1){
78.             require(keccak256(abi.encodePacked(value,guessvalue)) == commitments[player1],
     "Incorrect hash value");
79.             valuesP1P2 storage valuesPlayer = Fvalues[1] ; // save values of player1
80.             valuesPlayer.value = value;
81.             valuesPlayer.guessvalue = guessvalue;
82.             value1 = value;
83.             valueguess = guessvalue;
84.             revealP1= true;
85.
86.         }else if (msg.sender == player2){
87.
88.             require(keccak256(abi.encodePacked(value,guessvalue)) == commitments[player2],
     "Incorrect hash value");
89.             valuesP1P2 storage valuesPlayer = Fvalues[2] ; // save values of player1
90.             valuesPlayer.value = value;
91.             valuesPlayer.guessvalue = guessvalue;
```

```
92.             value2 = value;
93.             valueguess2 = guessvalue;
94.             revealP2= true;
95.          }
96.
97.     }
98.
99.      function Play() external {
100.        require(revealP1 && revealP2 , "Reveal numbers or reset game");
101.        require(player1 != address(0) && player2 != address(0),"Provide both players");
102.
103.
104.        if (valueguess == value2 && value1 != valueguess2){//player1 wins
105.            winner = player1; //balance cant be accumulated from previous games
106.            loser = player2;
107.            prize = value1.add(valueguess);//+ valueguess;
108.            result = states[prize];
109.            balance[player1] = result; //address(this).balance; //prize; winner
110.            balance[player2] = (address(this).balance / 1000000000000000000).sub(result);

111.            gameOver = false;//
    can continue playing, i.e withdraw the prize and possible rebate
112.
113.        }else if (valueguess2 == value1  && valueguess != value2) {// player2 wins
114.            winner = player2;
115.            loser = player1;
116.            prize = value2.add(valueguess2);// + valueguess2;
117.            result = states[prize];
118.            balance[player2] = result; //winner
119.            balance[player1] = (address(this).balance / 1000000000000000000).sub(result);

120.            gameOver = false; //
    can continue playing, i.e withdraw the prize and possible rebate
121.
122.        } else if (valueguess2 != value1 && valueguess != value2 ||
123.                    valueguess2 == value1 && valueguess == value2)
    { // no one wins, each player takes their stake
124.            balance[player1] = 0;
125.            balance[player2] = 0;
126.            player1.transfer(stake);
127.            player2.transfer(stake);
128.            gameOver = true;
129.            can_reset = true;
130.        }
131.     }
132.
133.    function withdrawPrize() public {
134.        require(msg.sender == winner,"Only winner can withdraw.");
135.        require(!gameOver, "No winner or game ended, press reset to replay");
136.        balance[winner] = 0;
137.        msg.sender.transfer(result.mul( 1000000000000000000));
138.         if (loser == address(0)){ // there is no rebate for loser
```

```
139.             gameOver = true;
140.             can_reset = true;
141.         }else{
142.             losercan = true;
143.         }
144.     }
145.
146.     function withdrawRebate() public {
147.         require(msg.sender == loser,"Only loser might have a rebate.");
148.         require(address(this).balance > 0,"No rebate.");
149.         require(!gameOver, "The game ended, press reset to replay");
150.         require(losercan, "Winner must withdraw first or rebate has been pulled.");
151.         balance[loser] = 0;
152.         if (prize == 2)
153.             msg.sender.transfer(3 ether);
154.         else if (prize == 3)
155.             msg.sender.transfer (2 ether);
156.         else if (prize == 4)
157.             msg.sender.transfer (1 ether);
158.
159.         losercan = false;
160.         can_reset = true;
161.
162.     }
163.
164.      function Reset () external {
165.     require(can_reset == true, "Players must withdraw first");
166.        gameOver = false; //the game starts again
167.        can_reset = false;
168.        balance[player1] = 0;
169.        balance[player2] = 0;
170.       revealP2=  false;
171.       revealP1 = false;
172.        player1 = address(0);
173.        player2 = address(0);
174.        winner = address(0);
175.        loser = address(0);
176.        value1 = 0;
177.        valueguess= 0;
178.        value2= 0;
179.        valueguess2= 0;
180.
181.     }
182. }
183.
184. library SafeMath {
185.   function sub(uint256 a, uint256 b) internal pure returns (uint256) {
186.     assert(b <= a);
187.     return a - b;
188.   }
189.
190.    function add(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
191.         uint256 c = a + b;
192.         require(c >= a, "SafeMath: addition overflow");
193.
194.         return c;
195.     }
196.
197.     function mul(uint256 a, uint256 b) internal pure returns (uint256) {
198.           if (a == 0) {
199.             return 0;
200.         }
201.
202.         uint256 c = a * b;
203.         require(c / a == b, "SafeMath: multiplication overflow");
204.
205.         return c;
206.     }
207. }
```