Assignment 2

How to play

After deploying my contract, users have to choose a number to show, a number to guess and two passwords. After that, they have to generate the hashes (keccak256) of the following: *showCommit = hash(<number to show><password1>)* and guessCommit = hash(<guess number><password2>). They can generate the hashes using the *hash* function of the contract, however, in this case, they need to run a contract on the local virtual machine. I.e. My chosen number is 3, my guess number is 2, my first password is pass1, my second password is pass2. In this case, my showCommit value is the hash of "3pass1" and my guessCommit value is the hash of "2pass2".

After that, they have to commit these values using *commit* function. It is important to commit the hash results (*showCommit* and *guessCommit*), and not the numbers or the passwords. They also have to pay 6 ETH deposit via this function.

After both of the users have committed their hashes, they have to reveal their numbers and passwords. They have to use *reveal* function by passing *<number to show><password1>* and *<guess number><password2>* values as arguments. I.e. I have to pass "3pass1" and "2pass2" as arguments.

After both users have revealed, the game is evaluated. The winner gets (almost) all his ETH back plus their prize. The looser gets back (almost) their balance minus the prize the winner won. To be as fair as possible, the price of the evaluation function (game) is halved and the player who executed this function gets plus prices from the other player.

If the opponent does not answer, a player has the ability to get all their money back after 30 seconds by using the *timeout* function. In this way, the user wins all of the deposited money of the opponent (if any).

High-level decisions

- Commit-reveal strategy: The smart contract runs on the blockchain. That is why if a function receives a parameter or saves data to the blockchain, it could be seen for the nodes, so an unfair player would see the chosen numbers. That is why the users are required to commit the hashes of their numbers and passwords. After they committed them, they reveal them. Since it is very unlikely to find another number+password pair which generates the same hash value, it cannot be predicted which are the numbers (showed and guess numbers) after the commitment but can be easily detected after revealing if the players changed their numbers.
- Deposit amount: In order to prevent cheating, users are required to send 6 ETH to the contract when they commit their numbers. This deposit is necessary for playing. The amount of deposit has been defined this way because of two main reasons. Firstly, if they had to transfer as much ETH as their showed number, the adversary would know the number. In addition, they might have sent the wrong amount of ETH which would be identified only during revealing. Secondly, the amount must be more than the maximum amount of ETH they can loose, which is 5. That is why the necessary amount is 6 ETH. This rule, unfortunately, disadvantages all the users who would like to play but have less than 6 ETH on their account.

- Transactions: After each player revealed their numbers, the game is evaluated. If only A wins, the necessary amount of ether is sent to A, if B wins, the necessary ether sent to B. In case of a draw (when no one or both of them win) they receive the deposited ethers back. The necessary ether is evaluated in the following way: their balance is 6 after the deposit. If A wins, win prize (i.e. 2) is transferred to A's balance (balance of A: 6 + 2 = 8, balance of B: 6 - 2 = 4). To be as fair as possible, a necessary amount of ETH (i.e. x) is transferred to the balance of the player who is executing the evaluation function (i.e. B executes the function: balance of A: 8 - x, balance of B: 4 + x). The calculation of x will be explained under Gas evaluation section. After that, their balance is transferred to them with the use of transfer function, to be as safe as possible. In order to prevent further attacks, their data is deleted from the blockchain before the transfer occurs.
- Preventing deadlock: Deadlock could occur if an adversary starts a game by committing values and then does nothing else. (The reason for it could be that the unfair player checked the revealed numbers of their opponent, and knew that the opponent would win.) In order to prevent this, two decision was made: (1) a given amount (6) of ETH has to be paid during commitment (as stated before) and (2) after a given time (30 seconds) timeout function can be called and the "good" user gets all the deposited amounts (both their and the attacker's deposit). In this way, the unfair player loses 6 ETH, which is more than any other cases.
- Timeout function: as it is mentioned below, timeout function aims making the user be able to withdraw from the game. After 30 seconds of inactivity, the user can call this function. Then it is checked if the user is in the game and if the user has taken further steps than the other user. If the user has made a commitment and there is no other user, the user gets their deposit back. If the user has revealed their numbers and the other one has committed, the user (who has revealed) gets the whole deposits back (even the deposit of the other user). In both ways, all data of the users are deleted.
- The number of players: In order to make the contract as simple as possible, the game can be played only with two players simultaneously.
- End of game: after both players have revealed their numbers, the game is automatically evaluated (*game* function). After that, all data is deleted about the players and the necessary money is sent back to them. After that, a new game can be started in the same contract.

Gas evaluation

The Table below shows the costs of deploying and interacting with my contract. These results were measured by Remix using JavaScript VM. Having tested my contract on the private blockchain, the costs were about the same.

	Transaction cost	Execution cost
Deploying	1 637 933 gas	1 212 861 gas
First commit	129 853 gas	104 229 gas
Second commit	131 789 gas	106 165 gas
First Reveal	47 294 gas	23 846 gas

	Transaction cost	Execution cost
Second Reveal* (see detailed explanation and solution below)	63 177 gas	102 905 gas
Timeout (after commit)	39 142 gas	57 011 gas
Timeout (after reveal)	50 131 gas	78 989 gas

In order to reduce gas costs, different steps have been done. Apart from keeping the contract as simple as possible, I used the cheapest hash function (*keccak256*), chose the order of my struct fields carefully [5], used require function instead of assert function [2], memory instead of storage and a predefined length array (*players*) [3]. These strategies reduced the necessary costs of my contract. I also tried to sort the functions to have the most popular (*commit*) in the beginning of my code in order to reduce the gas cost of this function [2].

My contract tries to be as fair as possible. The cost of hash, timeout and commit functions are about the same for all users as it can be seen in the table above. A small difference can occur if the beginning of the *commit* function has to check more conditions.

During the development, the main problem with fairness was the evaluation of the game. Since the evaluation runs only once per game (after both of the players have revealed), the player who called the reveal function secondly had to pay for the evaluation (*game*) function, as it can be seen in the table too. In order to make this fair, the amount of gas is measured during evaluation process (*game* function). After that, half of this amount is transferred to the balance of the executer from the balance of the other player. After that, these amounts are sent to the players as well as the other prizes and deposits which were mentioned before. In this way, the contract is more fair since the second player has to spend about the same as the first player. However, the fairness is still not perfect since the executer has to pay the fees of the transfers.

In order to make the contract more fair, it would be possible to precompute the necessry gas for *game* function, and half of the amount would be automatically transferred to the balance of the second player who executes the function. It would be also possible to create a public evaluate function. This function could be used to evaluate the game and send the necessary amount to the executer only, so both of the players should run this function. However, then it should be saved who has run this function already in order to prevent sending a prize again and again to a user.

The gas prices of the contract could be reduced by using struct encoding [5]. It would be also possible to store smaller datas such as the showed and guessed number as well as status in a single number, where different digits would represent different information. The fees would also be smaller in case of reducing the security: without a commit-reveal scheme the contract would be much simple, however, much vulnerable too.

Potential hazards and vulnerabilities

- Precomputed hashes: There is a low possibility that one could find two number+password pairs which both start with a number from 1 to 5 and the hashes of them are equal (i.e. let us say that someone has found out that 1asd and 3bsd have the same hash value). However, in this case, they could use them in an unfair way: committing the hash (which is the same), then waiting for the revealing of the opponent

(which can be seen from the blockchain), and then revealing the number + password pair which is the better for the adversary. I.e. if the opponent revealed 1 as the showed number, the adversary could say that “my guess was 1 and my password was asd”. In order to prevent this attack, the smart contract could generate a random string for every new game (i.e. based on the addresses of the players and the id of the current block), and would require the users to use this as part of the password (i.e. committing the hash of the following: <number><predefined word><passwords>). In this way, the committed value would be still safe (so one would be unable to guess the numbers before revealing) and the adversary would not be able to generate numerous hashes in advance to find collisions.

- Bad use of the contract: If the user forgets that the hash function should be used in their local VM, the parameter of the hash would be seen for an adversary. To prevent this, users should always use this function on the local VM.
- Revealing too fast: If the player has committed their values, they may try to revealing before the commitment of the other user. In this way, their revealing is not accepted, however, the adversary may be able to see their revealed parameters so could choose their numbers according to these. To prevent this (1) users should reveal their numbers after 30 seconds of their commitment. If the revealing is not accepted (so the other user did not commit yet), the user could call timeout and get their deposit back. Or (2) events should be used in order to notify users about the changes in the contract (i.e. ‘X has just committed, you can reveal now’).
- Changing *block.timestamp*: my contract uses *block.timestamp* in order to decide if the timeout has already occurred. However, this timestamp could be changed by the miner of the block. In this way, an attacker could reveal their number fast, then mining a block, modifying its timestamp in order to be able to call timeout function sooner than 30 seconds and “win” 6 ETH. However, “[p]opular Ethereum protocol implementations Geth and Parity both reject blocks with timestamp more than 15 seconds in future” [4]. In this way, the opponent still has at least 15 seconds (since the timeout is 30 seconds) to reveal their numbers. In order to mitigate this attack, users should be warned to reveal their numbers as fast as possible, or the timeout second should be modified (i.e. to have 60 seconds for revealing and committing).

Analysis of the contract of a fellow student

The code of my fellow student (Cedric Ecran) can be seen below under Code of contract of my fellow student. The contract works well in terms of the rules of Morra (winning and losing money).

However, the contract cannot be considered as safe. The players have to send their numbers (chosen showed number and guess number) to the contract without any encryption. In this way, an adversary could see the numbers of the user and could easily win the game by sending the necessary numbers to the contract. I. e. player A choose 3 and guess 2, and sends it to the contract. Then these values can be seen from other nodes of the blockchain, so anyone could send i.e. 1 as a chosen and 3 as a guess number. In this way, the adversary could always win.

In addition, a user can play with themselves. However, in case of a draw, only the first player receives their money back, the amount of deposit of the second player disappears.

In terms of fairness, the distribute function is unfair. It can be run only once in a game, so one of the players have to pay the cost of this function. There are some bad coding

practices in the program which are not necessary but make the gas costs higher, such as using a variable which is always 0 (*distribute* function: *deposit* variable) and initialising a variable to 0 (*deposit* variable). Using *enum* instead of a *uint* can also more difficult and less efficient but makes the code more readable.

The transaction history

A transaction history of a game with my contract.

	Transaction id	Activity log
Deploying the contract	0x552a03a2430d21e38253743298e9a327f3a721acff66a4062d6f980b642762d8	Transaction created with a value of 0 ETH at 20:39 on 11/1/2020. Transaction submitted with gas fee of 0 WEI at 20:39 on 11/1/2020. Transaction confirmed at 20:39 on 11/1/2020.
Player 1 commit	0xb216660da9476e881e8f96f678c3b944f2d6724a5692e3562b8c09b8c45e9a2e	Transaction created with a value of 6 ETH at 20:58 on 11/1/2020. Transaction submitted with gas fee of 0 WEI at 20:58 on 11/1/2020. Transaction confirmed at 20:59 on 11/1/2020.
Player 2 commmit	0xbd619bc002302888b1e72e657dd07b09cd8158a27fa8b6dba314557b6717b076	Transaction created with a value of 6 ETH at 24:59 on 11/2/2020. Transaction submitted with gas fee of 132989 GWEI at 24:59 on 11/2/2020. Transaction confirmed at 24:59 on 11/2/2020.
Player 1 reveal	0xfc97da9e50e673787d6a571489b4a805697d2b700b350a93c73ed04b07b927a2	Transaction created with a value of 0 ETH at 21:00 on 11/1/2020. Transaction submitted with gas fee of 0 WEI at 21:00 on 11/1/2020. Transaction confirmed at 21:00 on 11/1/2020.
Player 2 reveal	0x3d8409c500bad4821776f1374b6c0ffe74eed02298ffbaf6fad3f0d9ef1749fd	Transaction created with a value of 0 ETH at 01:00 on 11/2/2020. Transaction submitted with gas fee of 152535 GWEI at 01:00 on 11/2/2020. Transaction confirmed at 01:00 on 11/2/2020.

The time dates in activity logs are in different time zones.

References

[2] https://medium.com/layerx/how-to-reduce-gas-cost-in-solidity-f2e5321e0395

[3] https://medium.com/coinmonks/gas-optimization-in-solidity-part-i-variables-9d5775e43dde

[4] https://consensys.net/blog/blockchain-development/solidity-best-practices-for-smart-contract-security/

[5] https://medium.com/@novablitz/storing-structs-is-costing-you-gas-774da988895e

[6] https://mudit.blog/solidity-gas-optimization-tips/

Code of my contract

```
pragma solidity  >=0.4.22 <0.7.0;

contract morra {
    struct Player {
        uint8 showNum;
        uint8 guessNum;
        uint8 status; // 0: nothing, 1: committed, 2: revealed
        address payable addr;
        uint bal; // balance
        uint time; // last updated time
        bytes32 showCommit; // hash of <showedNumber><password1>
        bytes32 guessCommit; // hash of <guessNumber><password2>
    }

    Player [2] players;

    function commit(bytes32 showCommit, bytes32 guessCommit) payable public {
        // Decide if therre is free place in the game
        uint playerNum;
        if (players[0].status == 0){
            playerNum = 0;
            players[0].addr = msg.sender;
            players[0].status = 1;
        }
        else if (players[1].status == 0 && players[0].addr != msg.sender){
            playerNum = 1;
            players[1].addr = msg.sender;
            players[1].status = 1;
        }
        else {
            revert("No free place in the game");
        }
        // Check payed value and save values
        require(msg.value == 6 ether, "Player has to pay 6 ETH to play.");
        players[playerNum].bal = msg.value;
        players[playerNum].showCommit = showCommit;
        players[playerNum].guessCommit = guessCommit;
        players[playerNum].time = block.timestamp;
    }
```

```
function reveal(string memory showPass, string memory guessPass) public{
        // Check if user is in the game
        uint playerNum;
        if (players[0].addr == msg.sender){
            playerNum = 0;
        }
        else if (players[1].addr == msg.sender){
            playerNum = 1;
        }
        else{
            revert("Player is not on the game");
        }

        // Check if players have committed, the and validate inputs
        require(players[0].status >= 1 && players[1].status >= 1, "Some of the players
has not committed yet");
        require(players[playerNum].status == 1, "Player has already revealed");
        require(keccak256(abi.encodePacked(showPass)) == players[playerNum].showCommit,
"Showed number hash error");
        require(keccak256(abi.encodePacked(guessPass)) ==
players[playerNum].guessCommit, "Guessed number hash error");

        // Get numbers from passwords, validate numbers
        uint8 showNum  = uint8(bytes(showPass)[0])-48;
        uint8 guessNum  = uint8(bytes(guessPass)[0])-48;
        require(showNum > 0 && showNum < 6 && guessNum > 0 && guessNum < 6, "Showed or
guessed number error");

        // Save values
        players[playerNum].showNum = showNum;
        players[playerNum].guessNum = guessNum;
        players[playerNum].status = 2;
        players[playerNum].time = block.timestamp;

        // If both player have revealed, evaluate the game
        if (players[0].status == 2 && players[1].status == 2){
            game();
        }
    }
```

```
    function game() private {
        // To check how much gas this function requires
        uint startGas = gasleft();

        // Permamently save values to evaluate the game
        bool win0 = players[0].guessNum == players[1].showNum;
        bool win1 = players[1].guessNum == players[0].showNum;
        uint num0 = players[0].showNum * 1 ether;
        uint num1 = players[1].showNum * 1 ether;
        uint eth0 = players[0].bal; // ETH to send to player0
        uint eth1 = players[1].bal; // ETH to send to player1
        address payable addr0 = players[0].addr;
        address payable addr1 = players[1].addr;

        // Delete values on Blockchain
        delete players[0];
        delete players[1];

        // Decide who won
        if (win0 && !win1){
            eth0 +=  num1;
            eth1 -= num1;
            require(eth0 >= 6 ether && eth1 < 6 ether); // check overflow
        }
        else if (win1 && !win0){
            eth1 +=  num0;
            eth0 -= num0;
            require(eth1 >= 6 ether && eth0 < 6 ether); // check overflow
        }

        // Check how much gas was used
        uint gasUsed = startGas - gasleft();
        // Split the gas used in this function in order to be as fair as possible
        if (msg.sender == players[0].addr){
            uint pre0 = eth0;
            uint pre1 = eth1;
            eth0 += gasUsed/2;
            eth1 -= gasUsed/2;
            require(eth0 > pre0 && eth1 < pre1);
        }
```

```
        else{
            uint pre0 = eth0;
            uint pre1 = eth1;
            eth1 += gasUsed/2;
            eth0 -= gasUsed/2;
            require(eth1 > pre1 && eth0 < pre0);
        }

        // Send the counted ETHs to the users.
        addr0.transfer(eth0);
        addr1.transfer(eth1);
    }
    function timeout() public {
        // Check if user is in the game and if timeout occur
        require(players[0].addr == msg.sender || players[1].addr == msg.sender, "Player
is not in a game");
        require(block.timestamp > players[0].time + 30 && block.timestamp >
players[1].time + 30, "Timeout not reached");

        // Save values permanently
        uint256 eth0 = players[0].bal;
        uint256 eth1 = players[1].bal;
        uint status0 = players[0].status;
        uint status1 = players[1].status;
        address payable addr0 = players[0].addr;
        address payable addr1 = players[1].addr;

        // Delete data on blockchain
        delete players[0];
        delete players[1];
        // Send all ETH to the user who has done more steps
        if (status0 > status1){
            addr0.transfer(eth0 + eth1);
        }
        else if (status1 > status0){
            addr1.transfer(eth0 + eth1);
        }
        else{
            revert("Players are in the same status");
        }
```

```
    // generate the hash of a given string
    function generateHash(string memory word) pure public returns (bytes32){
        return keccak256(abi.encodePacked(word));
    }
}
```

Code of the contract of my fellow student

```
pragma solidity >0.5.0 <0.7.0;
contract Morra {

    enum Choice {
        None,
        One,
        Two,
        Three,
        Four,
        Five
    }
    enum Guess {
        None,
        One,
        Two,
        Three,
        Four,
        Five
    }

    enum Stage {
        FirstCommit,
        SecondCommit,
        Distribute
    }

    struct CommitChoice {
        address playerAddress;
        Choice choice;
        Guess guess;
    }
    event Payout(address player, uint amount);
```

```
// State vars
    CommitChoice[2] players;
    Stage public stage = Stage.FirstCommit;



    function Play(Choice choice, Guess guess) public payable {
         // Only accept valid choices
        // Only accept valid choices
        require(choice == Choice.One || choice == Choice.Two || choice == Choice.Three
|| choice == Choice.Four || choice == Choice.Five, "invalid choice");
        require(guess == Guess.One || guess == Guess.Two || guess == Guess.Three ||
guess == Guess.Four || guess == Guess.Five, "invalid choice");

        uint playerIndex;
        if(stage == Stage.FirstCommit) playerIndex = 0;
        else if(stage == Stage.SecondCommit) playerIndex = 1;
        else revert("both players have already played");

        uint commitAmount = uint(choice) * 1e18;
        require(msg.value == commitAmount, "value must be equal to commit ammount");



        players[playerIndex] = CommitChoice(msg.sender, choice, guess);


        if(stage == Stage.FirstCommit) stage = Stage.SecondCommit;
        // Otherwise we must already be on the second, move to first reveal
        else stage = Stage.Distribute;
    }
```

```
    function distribute() public {
        // To distribute we need:
            // a) To be in the distribute stage OR
            // b) Still in the second reveal stage but past the deadline
        require(stage == Stage.Distribute, "cannot yet distribute");

        // Calculate value of payouts for players
        uint player0Payout;
        uint player1Payout;
        uint deposit;
        uint winningAmount = (uint(players[0].choice) + uint(players[1].choice)) *
1e18;
        deposit = 0;



        // If both players picked the same choice, return their deposits and bets
        if((uint(players[0].guess) == uint(players[1].choice)) &&
(uint(players[1].guess) == uint(players[0].choice))) {
            player0Payout = uint(players[0].choice)* 1e18;
            player1Payout = uint(players[1].choice)* 1e18;
        }
        else if(uint(players[0].guess) == uint(players[1].choice)) {
            assert(uint(players[1].guess) != uint(players[0].choice));
            player0Payout = winningAmount;
            player1Payout = deposit;
        }
        else if(uint(players[1].guess) == uint(players[0].choice)) {
            assert(uint(players[0].guess) != uint(players[1].choice));
            player0Payout = deposit;
            player1Payout = winningAmount;
        }
        else {
            player0Payout = uint(players[0].choice)* 1e18;
            player1Payout = uint(players[1].choice)* 1e18;
        }
        // Send the payouts
        if(player0Payout > 0) {
            (bool success, ) = players[0].playerAddress.call.value(player0Payout)("");
            require(success, 'call failed');
            emit Payout(players[0].playerAddress, player0Payout);
```

```
        } else if (player1Payout > 0) {
            (bool success, ) = players[1].playerAddress.call.value(player1Payout)("");
            require(success, 'call failed');
            emit Payout(players[1].playerAddress, player1Payout);
        }

        // Reset the state to play again
        delete players;
        stage = Stage.FirstCommit;
    }
}
```