# Blockchains & Distributed Ledgers - Coursework 2

s1519734

April 18, 2025

## 1 High-level decisions

I will be going through the high level decisions I have taken by following how a typical game of Morra would be played using my contract.

### 1.1 General considerations

In my implementation, only two players can play Morra at any given time. My contract does not support parallel games.

### 1.2 Game start - commit phase

Each player (px) starts by picking the number he is going to play (nx) and a guess (gx) for the other player's number. Both nx and gx have to be between 1 and 5 (inclusive) in order for them to be valid.

Then, the player calculates the number NGx = nx*8 + gx (relevant for saving gas, see Section 3).

The player then needs to compile the "compute_hash" function (see Section 4) in a VM and input the address of the account he will be playing the game from (in the "player" field), the number NGx calculated previously (in the "played_nr_and_guess") and a random number which should not be larger than $2^{256} - 1$ (in the "safety_number" field) - the player should save this number. This function packs those arguments together, hashes the result using keccak256 and finally emits the hash. Those choices will be justified in Section 2. At a high level, I can say that the game will be played using a commit reveal scheme, so that no player will be aware of what the other has played.

Then, each played needs to call the "commit" function with the hash they have produced as input. They should also include a 6 ether deposit with this transaction (see Section 2.3).

Once the first player has committed, there is no time limit for a second player to commit. If the first player decides that he waited too long, he can cancel his participation (by calling the "cancel_participation" method). Providing that a second player has not committed his hash before this function was called, the player gets 5 ether back (using pull over push, see

Section 2.4). By withdrawing from the game, the player loses his deposit (1 ether). The reasoning for this is that I wanted to discourage this player behaviour, while not making it impossible for a player to withdraw from the contract. The penalty could be adjusted based on the current price of ETH (in the real world a 1 ETH penalty would be quite harsh).

## 1.3 Reveal phase

Once both players have committed their hashes, they cannot withdraw from the game anymore (the reveal phase begins). This serves as an incentive for them to play the game normally and get their rewards.

Each of the two players needs to call the "reveal" function, with the NGx and the random number which were used to generate their hash in the commit phase. For the reveal to be successful, several things need to happen:

1. The message sender needs to be one of the two players who have committed.
2. Both players should have committed beforehand.
3. The reveal should happen in a block which is within a number of blocks from the block in which the last commit happened. This number of blocks is represented by the "grace_period" constant in our code (currently 30 blocks). On the public ETH chain, 4 blocks are mined per minute (I went with this number as an approximation for the number of blocks mined per minute in our private chain). A grace period of 30 blocks would give both players roughly 7 minutes to reveal their picks.
4. The reveal function computes the hash using the sender's address and the two input numbers. If the hash matches the one which was committed by the player in the commit phase, then the reveal is successful and NGx is saved in the storage.

Once both players have successfully revealed or the grace period has passed, the reveal phase is over.

## 1.4 Deciding the winner

If both players have successfully revealed their NGx, one of them can call the "decide_winner" function. This method first derives nx and gx for each player from NGx (the three rightmost bits of NGx represent gx and the next three bits to the left represent nx). If all those numbers are valid (they are between 1 and 5 inclusive):

- If g1 == n2 and g2 == n1 (both players have guessed correctly) or g1 != n2 and g2 != n1 (both players have guessed incorrectly), then the game is considered to have ended in a draw. Thus, both players get their currency back (6 ether).
- If only one player guesses correctly (g1 == n2 and g2 != n1 or viceversa), then the player who has guessed correctly is declared the winner. 6 ether + (the number picked by the loser) x 1 ether is added to the winner's balance and 6 ether - (the number picked by the loser) x 1 ether is added to the loser's balance.

Thus, each player stands to lose an amount of ether equal to the number he has played and stands to win an amount of ether equal to the number played by the other player (as required by the problem statement).

If only one player had valid values for nx and gx, then he is awarded the maximum number that can be picked by the other player in ether (so, 5 ether). The player who was submitted invalid values is considered to have lost. Furthermore, since his hash was invalid, he will lose his deposit, which can be burned or given to the contract owner.

If none of the two players submitted valid values for nx and gx, then both players lose their deposit (1 ether each), and 5 ether is added to their balance.

Finally, the function resets the state of the game.

Perhaps it is important to mention that if this function is not called by either player within a number of blocks (equal to the grace period mentioned earlier, 30 blocks) from the last reveal, then the contract owner can end the game himself (see the "forcefully_end_game" function).

## 1.5 Withdrawing the outstanding balance

After the game is over, each player can call the "withdraw_balance" function. The function retrieves the caller's balance, saves it in a temporary variable, zeroes its storage value and then transfers this outstanding balance to the caller. Each player's balance is stored in an "address => uint256" mapping, in storage.

## 1.6 Corner cases

There are a few corner cases which I have not addressed yet:

1. None of the two players successfully reveal during the grace period.
2. Only one player successfully reveals during the grace period.
3. Both players successfully reveal, but none of them call the "decide_winner" function within the grace period (from the last reveal).

If any of those corner cases happens, the contract owner must intervene and call the "forcefully_end_game" function (which can be called only by the contract owner). The method checks that the game state corresponds to one of the three corner cases and acts as follows, for each corner case:

1. For corner cases 1 and 3, both players lose their deposit of 1 ETH each, and the rest (5 ETH) is added to each player's balance.
2. For corner case 2, the player who has revealed gets the maximum amount of winnings (5 ETH), while the player who hasn't loses all his 6 ETH. The deposit (1 ETH) is either burned or added to the contract owner's in-contract balance.

If this contract were to be deployed on a public chain, I would have also emitted events after each reveal and commit. As a contract owner, I would be able to subscribe to the logs emitted by my contract and I would detect whenever such a corner case occurs and call this method.

For a private chain, the owner can create a (non-Solidity, external) script to call the ill-named "forcefully_end_game" every n hours and he can adjust this number based on how many times the method executes (aka: how many times a corner case is encountered, since those are the only cases when the function actually ends the game).

Interestingly enough, a player can play the game with himself. This is not explicitly disallowed, since one such player will lose around 400K gas per game.

The owner of the contract can also play the game, as he has no inherent advantage over an honest player.

### 1.7 Other considerations

Whenever a player breaks the rules, his deposit (1 ether) is added to the account owner's balance. This is just so no ether is lost in our private blockchain, but it may not be perceived as fair by the players. This balance can just as easily be burned instead (sent to 0x). However, the contract owner also needs an incentive to keep monitoring the state of the contract and call "forcefully_end_game" whenever needed. So ideally, the owner should be given an incentive from the deposit while the rest of the deposit is burned or returned.

## 2 Hazards, vulnerabilities and security mechanisms

In this section, I will talk about the the hazards and vulnerabilities I have thought about while implementing my contract and what security mechanisms I adopted in order to mitigate them.

### 2.1 Commit-reveal

In my contract, the players submit their picked numbers and guesses using a commit reveal scheme.

**Motivation for using the scheme**
The alternative would have been for the players to submit their picked numbers and guesses as-is. The blockchain is public, so an opponent (in the case of our ETH blockchain, perhaps a miner?) can see what a player has picked and win the game trivially. Splitting the game into the two phases (commit and reveal) ensures that players cannot cheat in this way. The reveal phase starts only after the commit phase has been finished, and the commit phase binds a player to his choice. Assuming that the players have access to their opponents' committed hashes, one objective I had was to make reverse-engineering a players' hash impossible.

What I mean by this is that the picked number and guesses can be numbers between 1 and 5. Simply concatenating their bits and hashing the result would have yielded 25 possible valid hashes, making it easy for an opponent to figure out what the legitimate player picked and guessed. This is why my hash function concatenates the bits of the picked number and guess to a uint256 integer (the "safety_number" variable). This number should ideally be generated by the player using a randomness source that is independent from the blockchain.

## 2.2 Front-running

Preventing front-running is the reason why I included the address of the player who computed the commit hash into the hash as an input. This way, if a griefer manages to submit another player's hash before him, denying him access to the current round, the griefer will never be able to retrieve his 6 ether. Even though he will be registered as a player (and thus will be able to call the reveal function) and even though he may have been able to steal the player's NGx and safety number, he still wouldn't be able to perform a successful reveal (since his account address will be different than the address of the legitimate player and thus the hash computed in the reveal function will be different than his commit hash).

## 2.3 Why do the players need to submit 6 ether?

Each honest player stands to lose his picked number in ether, so at most 5 ether. If every player submitted his picked number in ether when committing, an opponent may see this transaction and infer the number played, making a win trivial. As a consequence, each participant should submit the same amount of ether, which must be greater than or equal to 5.

The difference between this submitted amount and 5 can be seen as a "deposit" of sorts, which serves an additional incentive for the player to have a nice behaviour: not to withdraw, submit his reveal in time and call decide_winner in time. If a player doesn't do any of these, he will surely lose his deposit (and depending on the actions of the other player in some cases, he may lose the 5 ether too). In my implementation, the deposit is equal to 1 ether.

## 2.4 Pull over push

Source: `https://fravoll.github.io/solidity-patterns/pull_over_push.html`.
Pull over push is a pattern which shifts the responsibility of transferring ETH from the contract to the user. The pattern requires the contract to have a mapping from the player's addresses to their balances ("players_balances" in my contract) and a withdraw function, which is public and can be called by any player in order to withdraw his outstanding balance.

The pattern isolates the transfer logic from the rest of the contract. The alternative would have been to send the funds in the decide_winner function. What could have happened if I did that: one of the players may have actually been a contract. If it were malicious, the transfer could trigger its fallback function, which could consume the rest of the available gas. As a result, the other player could be denied from his rewards, albeit temporarily. (griefing) Its fallback function may also call decide_winner again, opening up the possibility of a reentrancy attack depending on the order of the operations performed in decide_winner (aka when the players' data is deleted).

This pattern ensures that when a player withdraws his balance, no other present or past player is affected and also ensures that each player is responsible for withdrawing his balance.

### 2.5 Reentrancy attacks

The only function in my implementation which can be subject to a reentrancy attack is the withdraw_balance function. This is why I made sure that before sending the ether to the user who requests the withdrawal: 1) the user's balance is different from zero and 2) I make sure to update his balance to zero before sending the ether. This renders a reentrancy attack useless.

### 2.6 Integer overflow and underflow

Whenever an integer underflows or overflows, this can lead to unintended consequences (as illustrated in Lecture 4). The two methods in which integer overflow or underflow could theoretically occur are the decide_winner and forcefully_end_game functions.

The only way an integer overflow can occur is while adding ether to a player's in-contract balance. Unless there are currently more than $2^{256} - 1$ ether coins in our private blockchain, an integer overflow cannot occur (since my contract does not generate ether from thin air and does not offer to an attacker the opportunity to do so - hopefully).

However, due to an error on my part, an integer underflow can currently occur in the decide_winner function, while calculating how much gas was consumed (near the bottom of the method). The formula in the currently deployed contract is: "uint256 gas_consumed = (gas_beginning - 40000 - gasleft()) * tx.gasprice;". If the gas at the beginning of the contract execution - gasleft() is lower than 40000, this will lead to an underflow. Multiplying by tx.gasprice will then likely lead to an overflow. The simple fix is to simply remove the "-40,000" term. I included it because I wanted to take into account the gas the player who called the function gets back as a result of deleting storage variables, but this was not the correct way to do so. This issue has been fixed in the submitted code.

## 3 Gas evaluation

### 3.1 Gas optimisation objectives

After finishing my code, I had to make several modifications in order to optimise its gas usage. My objectives (in decreasing order of importance) were:

- Minimising the cost of interacting with the contract (for the user).
- Making sure that the gas usage is fair to both players (aka one of the player doesn't have to pay significantly more gas than the other).
- Minimising the cost of deploying the contract.

## 3.2 Techniques for cost effectiveness and fairness

### 3.2.1 Cost effectiveness

Source: https://ethereum.stackexchange.com/questions/35539/what-is-the-real-price-of-deploying -a-contract-on-the-mainnet. Every contract deployment has a standard fee. Additionally, each byte from the compiled bytecode of the contract will incur an additional cost. Therefore, there is an incentive to keep the contract code as small as possible.

Another thing that should be taken into consideration is the usage of storage variables. Setting one such variable from a zero value to a non-zero one incurs a fixed cost of 20,000 gas. Subsequent modifications to non-zero values cost 5,000 gas each. In-memory variables are much cheaper. Therefore, storage variables should be used only when it is completely necessary.

Another technique for minimising gas usage is variable packing (source: https://fravoll.github.io/solidity-patterns/tight_variable_packing.html). To optimise storage usage, we must group the variables we declare so that we minimise the number of 32-bytes storage slots used (if possible). Two storage variables declared one after the other and which have a total combined size of less than 32-bytes can be stored in the same storage slot.

The cost optimisations I made as a consequence of those observations:

- Removed extraneous variables. For example, instead of storing two integers for each player (one for the pick and one for the guess), I made use of the fact that each of those two integers can have at most 3 bits, and I stored both in the same integer (its last 3 bits representing the guess and the previous 3 bits representing the picked number).
  In a previous version of my code, I had a storage integer which was keeping track of the number of players who had committed. However, this number can be inferred by looking at the players' commit hashes, which is what I did.
  Similarly, I also had an integer which was keeping track of the number of players who had revealed, and which I now infer from players[player_address].revealed.
- I changed the order in which my storage variables were declared in order to reduce the number of storage slots used.

By reducing the number of variables associated with each player, the cost of running my contract's functions also went down (since they had to modify fewer such variables). Removing the extraneous variables was not trivial, as a lot of contract logic was relying on them. With these two optimisations, Remix' estimation of the contract deployment price went down by 600K gas. Similarly, a normal player interaction with the contract during a game of Morra went down from 300K gas to around 220K gas (again, using the gas estimations provided by Remix).

### 3.2.2 Gas fairness

During a normal game of Morra, each of the two players needs to call commit and reveal once and perhaps withdraw_balance too. Additionally, one of the players needs to call the decide_winner function.

For the commit and reveal functions, my aim was to make the gas costs similar for the two players. At the beginning, in the first contract execution, the first player who called either of those functions ended up paying around 15-20k gas more than the second player. This was due to the fact that he changed the values of the "last_commit_block" and "last_reveal_block" variables from zero to non-zero, thus incurring a higher cost than the next player, who would modify those from a non-zero value to another non-zero value. The simple fix is to initialise both of these values to a non-zero, small value (1) in the contract constructor. This way, the first player will also modify it from a non-zero value to a non-zero value. With this change, commit and reveal are symmetrical for both players, so they should incur a similar gas cost. Withdraw_balance is also the same for each player, so there's no problem there.

The main issue in terms of fairness is that only one player calls the decide_winner function, whose execution costs around 60k gas according to Remix. In order to remediate this, after most calculations in the function have been done, the function calculates the price of the gas consumed so far using the formula: gas_consumed = (gas at the beginning of the function call - gasleft()) * tx.gasprice. In the case that both players played valid numbers, gas_consumed/2 is taken from the other player's balance and added to the caller's balance. As an added security measure, this redistribution only happens if the cost of the consumed gas is less than 1 ETH (in the case that the caller set a really high gas price; if left unchecked, this can lead to cancelling the other's player reward or it can lead to underflow issues. If the amount is limited to 1 ETH, at most 0.5 ETH will be subtracted from the other player's balance - obviously not ideal, the threshold can be lowered if needed).

## 3.3 Transaction gas costs

The gas cost of deploying the contract was: 2,884,457.
Cost of my commit: 69787 gas.
Cost of my partner's commit: the same.
Cost of my reveal: 70240 gas.
Cost of my partner's reveal: 70498 gas.
Cost of decide_winner: 59436 gas.
Cost of withdrawing: 19350 gas (same for my partner).
As we can see, the costs for commit, reveal and withdraw are balanced for each player. The discrepancy in gas costs between the two players' interactions (caused by decide_winner) will be rectified using the players' balances. On average, each player will spend approximately 190k gas per match.

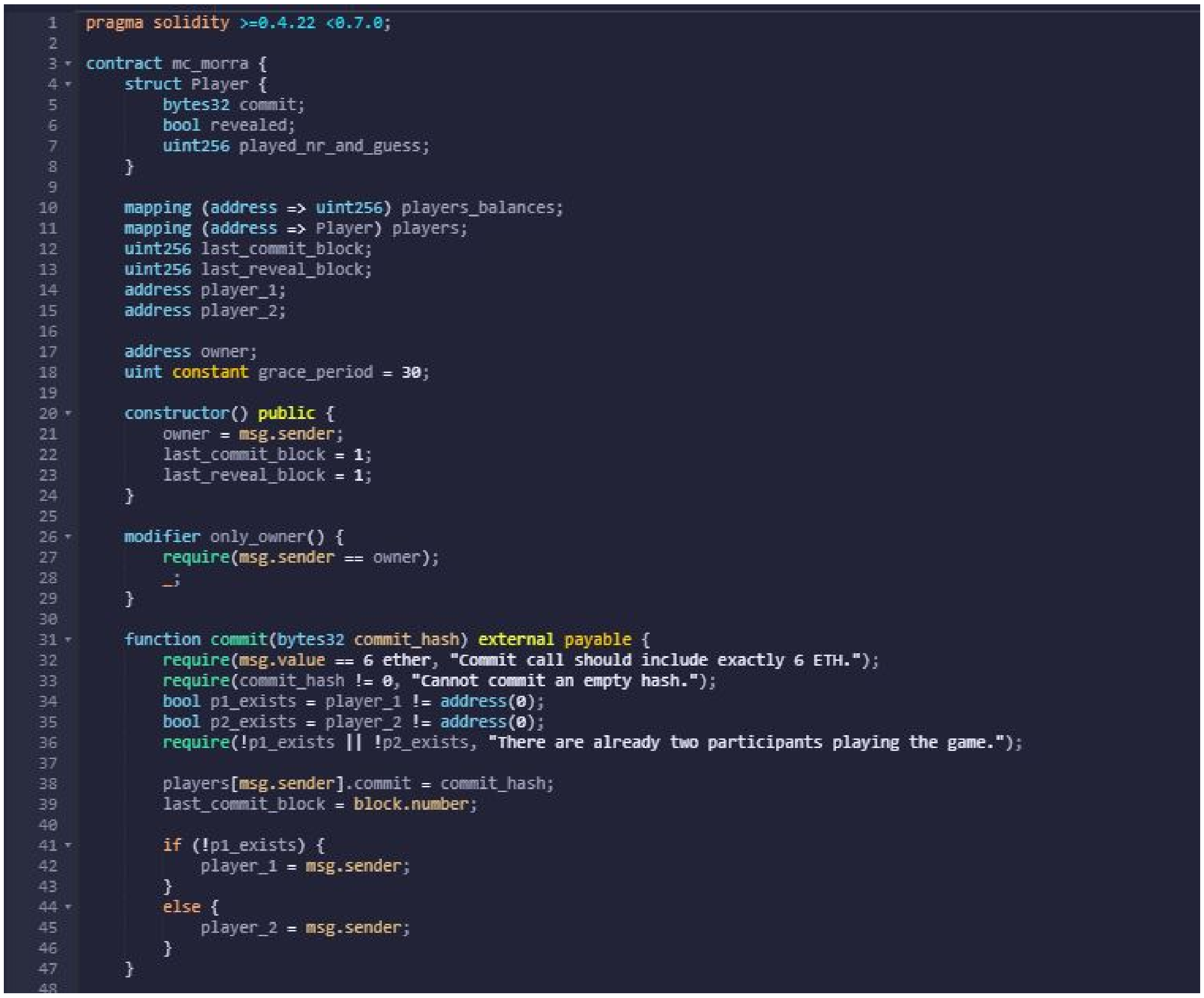
# 4 Contract code

After deployment, I have discovered and fixed the following bugs:

- Eliminated a useless pre-condition in the reveal function (I was checking whether the reveal happens in the same block as the one containing the last commit).
- Eliminated a potential underflow and overflow from decide_winner.

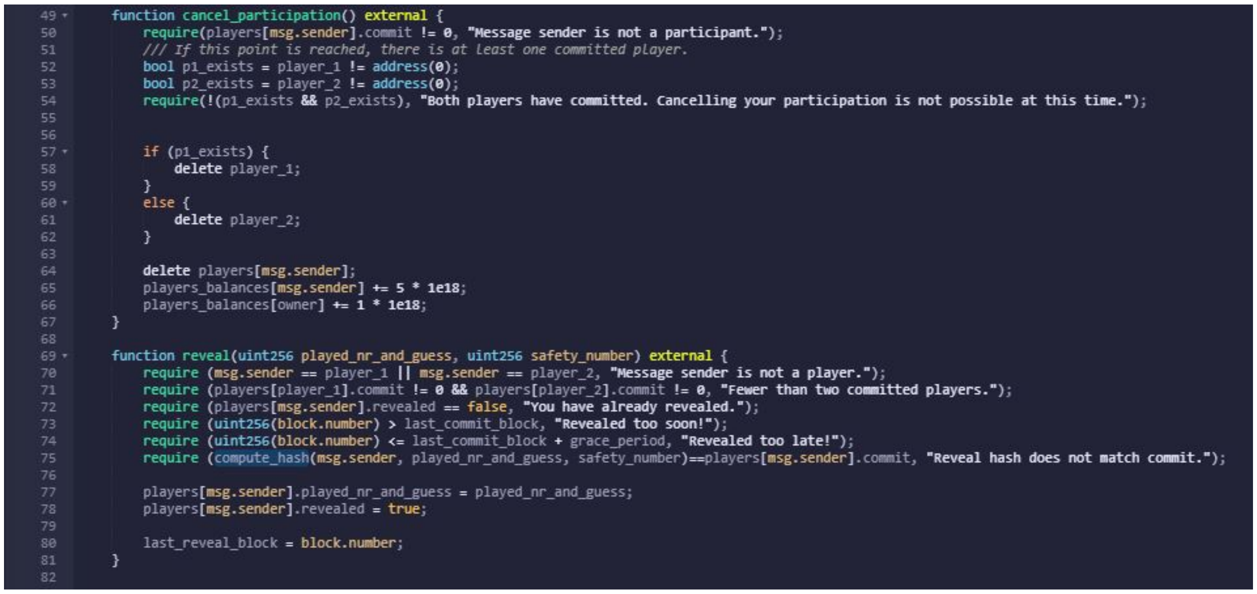
- Corrected some flawed logic in decide_winner: when only one player revealed valid numbers, the old code could give the gas refund to the incorrect player.

These bugs/flaws did not affect the game I played with my partner, so re-deployment was not necessary. The code below contains the fixes to those flaws.



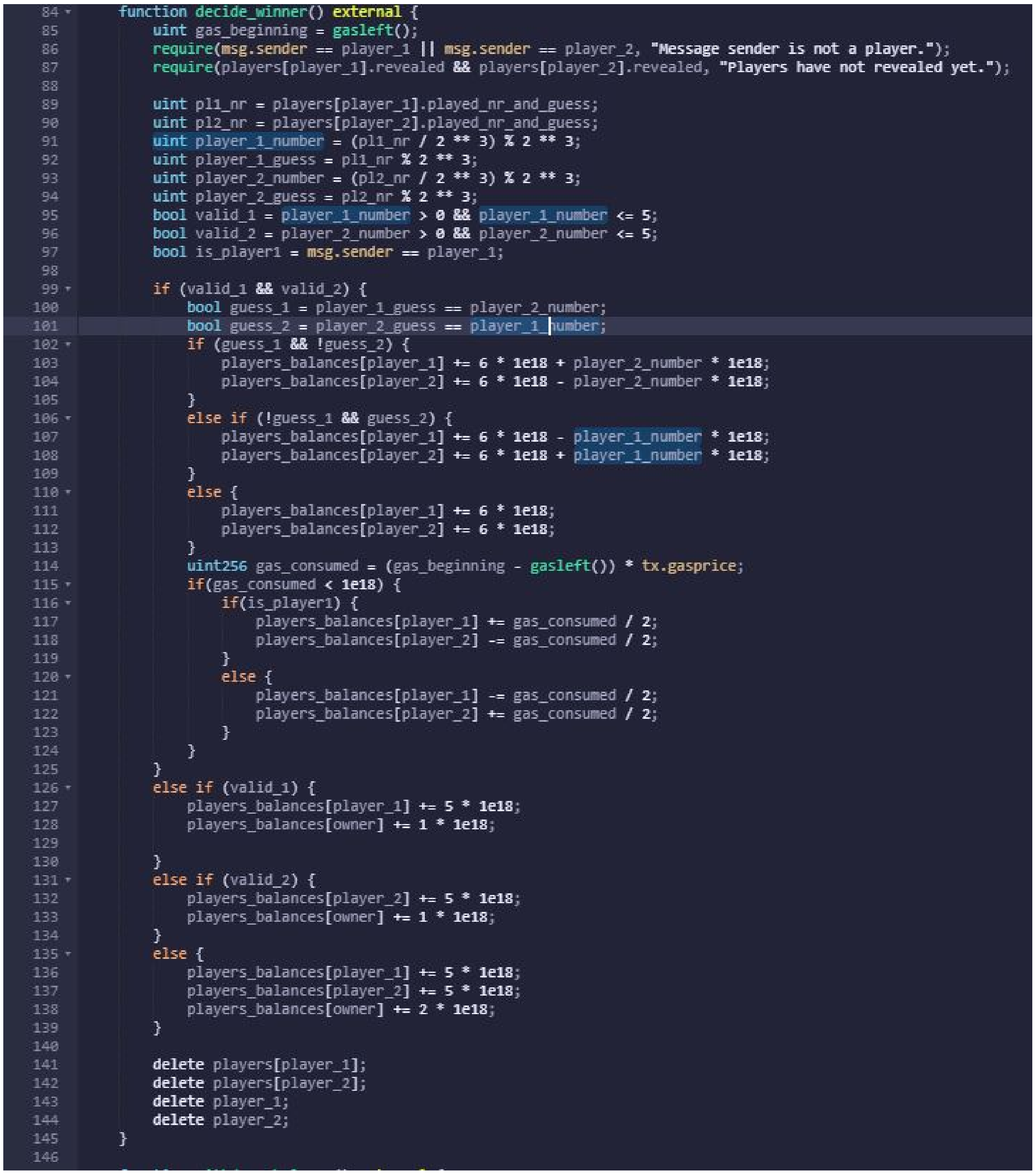
```
pragma solidity >=0.4.22 <0.7.0;

contract mc_morra {
    struct Player {
        bytes32 commit;
        bool revealed;
        uint256 played_nr_and_guess;
    }

    mapping (address => uint256) players_balances;
    mapping (address => Player) players;
    uint256 last_commit_block;
    uint256 last_reveal_block;
    address player_1;
    address player_2;

    address owner;
    uint constant grace_period = 30;

    constructor() public {
        owner = msg.sender;
        last_commit_block = 1;
        last_reveal_block = 1;
    }

    modifier only_owner() {
        require(msg.sender == owner);
        _;
    }

    function commit(bytes32 commit_hash) external payable {
        require(msg.value == 6 ether, "Commit call should include exactly 6 ETH.");
        require(commit_hash != 0, "Cannot commit an empty hash.");
        bool p1_exists = player_1 != address(0);
        bool p2_exists = player_2 != address(0);
        require(!p1_exists || !p2_exists, "There are already two participants playing the game.");

        players[msg.sender].commit = commit_hash;
        last_commit_block = block.number;

        if (!p1_exists) {
            player_1 = msg.sender;
        }
        else {
            player_2 = msg.sender;
        }
    }
```

Figure 1: Global variables, constructor and commit



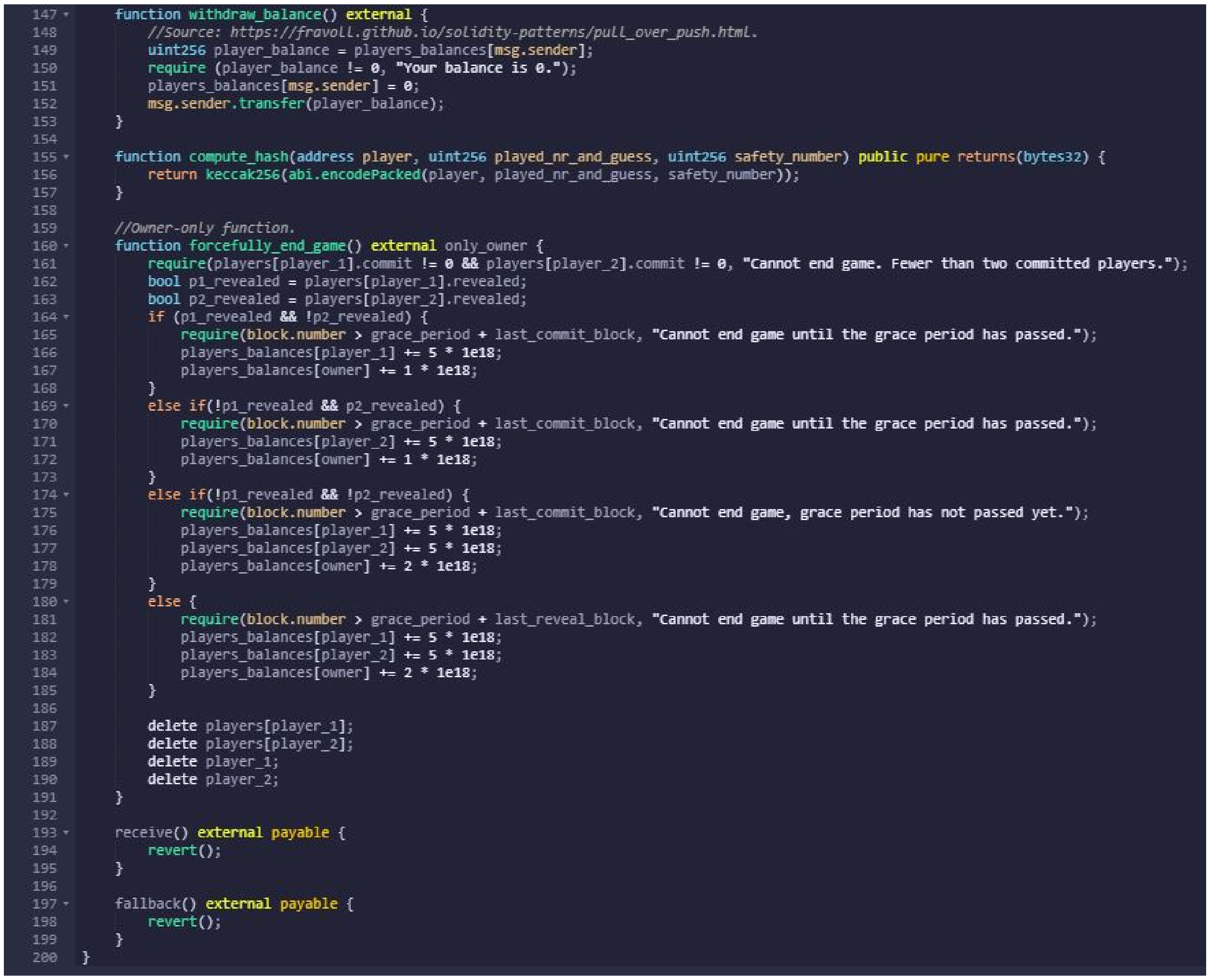
```solidity
    function cancel_participation() external {
        require(players[msg.sender].commit != 0, "Message sender is not a participant.");
        /// If this point is reached, there is at least one committed player.
        bool p1_exists = player_1 != address(0);
        bool p2_exists = player_2 != address(0);
        require(!(p1_exists && p2_exists), "Both players have committed. Cancelling your participation is not possible at this time.");


        if (p1_exists) {
            delete player_1;
        }
        else {
            delete player_2;
        }

        delete players[msg.sender];
        players_balances[msg.sender] += 5 * 1e18;
        players_balances[owner] += 1 * 1e18;
    }

    function reveal(uint256 played_nr_and_guess, uint256 safety_number) external {
        require (msg.sender == player_1 || msg.sender == player_2, "Message sender is not a player.");
        require (players[player_1].commit != 0 && players[player_2].commit != 0, "Fewer than two committed players.");
        require (players[msg.sender].revealed == false, "You have already revealed.");
        require (uint256(block.number) > last_commit_block, "Revealed too soon!");
        require (uint256(block.number) <= last_commit_block + grace_period, "Revealed too late!");
        require (compute_hash(msg.sender, played_nr_and_guess, safety_number)==players[msg.sender].commit, "Reveal hash does not match commit.");

        players[msg.sender].played_nr_and_guess = played_nr_and_guess;
        players[msg.sender].revealed = true;

        last_reveal_block = block.number;
    }

```

Figure 2: Cancel_participation and reveal



```
    function decide_winner() external {
        uint gas_beginning = gasleft();
        require(msg.sender == player_1 || msg.sender == player_2, "Message sender is not a player.");
        require(players[player_1].revealed && players[player_2].revealed, "Players have not revealed yet.");

        uint pl1_nr = players[player_1].played_nr_and_guess;
        uint pl2_nr = players[player_2].played_nr_and_guess;
        uint player_1_number = (pl1_nr / 2 ** 3) % 2 ** 3;
        uint player_1_guess = pl1_nr % 2 ** 3;
        uint player_2_number = (pl2_nr / 2 ** 3) % 2 ** 3;
        uint player_2_guess = pl2_nr % 2 ** 3;
        bool valid_1 = player_1_number > 0 && player_1_number <= 5;
        bool valid_2 = player_2_number > 0 && player_2_number <= 5;
        bool is_player1 = msg.sender == player_1;

        if (valid_1 && valid_2) {
            bool guess_1 = player_1_guess == player_2_number;
            bool guess_2 = player_2_guess == player_1_number;
            if (guess_1 && !guess_2) {
                players_balances[player_1] += 6 * 1e18 + player_2_number * 1e18;
                players_balances[player_2] += 6 * 1e18 - player_2_number * 1e18;
            }
            else if (!guess_1 && guess_2) {
                players_balances[player_1] += 6 * 1e18 - player_1_number * 1e18;
                players_balances[player_2] += 6 * 1e18 + player_1_number * 1e18;
            }
            else {
                players_balances[player_1] += 6 * 1e18;
                players_balances[player_2] += 6 * 1e18;
            }
            uint256 gas_consumed = (gas_beginning - gasleft()) * tx.gasprice;
            if(gas_consumed < 1e18) {
                if(is_player1) {
                    players_balances[player_1] += gas_consumed / 2;
                    players_balances[player_2] -= gas_consumed / 2;
                }
                else {
                    players_balances[player_1] -= gas_consumed / 2;
                    players_balances[player_2] += gas_consumed / 2;
                }
            }
        }
        else if (valid_1) {
            players_balances[player_1] += 5 * 1e18;
            players_balances[owner] += 1 * 1e18;

        }
        else if (valid_2) {
            players_balances[player_2] += 5 * 1e18;
            players_balances[owner] += 1 * 1e18;
        }
        else {
            players_balances[player_1] += 5 * 1e18;
            players_balances[player_2] += 5 * 1e18;
            players_balances[owner] += 2 * 1e18;
        }

        delete players[player_1];
        delete players[player_2];
        delete player_1;
        delete player_2;
    }

```

Figure 3: Decide_winner



```
    function withdraw_balance() external {
        //Source: https://fravoll.github.io/solidity-patterns/pull_over_push.html.
        uint256 player_balance = players_balances[msg.sender];
        require (player_balance != 0, "Your balance is 0.");
        players_balances[msg.sender] = 0;
        msg.sender.transfer(player_balance);
    }

    function compute_hash(address player, uint256 played_nr_and_guess, uint256 safety_number) public pure returns(bytes32) {
        return keccak256(abi.encodePacked(player, played_nr_and_guess, safety_number));
    }

    //Owner-only function.
    function forcefully_end_game() external only_owner {
        require(players[player_1].commit != 0 && players[player_2].commit != 0, "Cannot end game. Fewer than two committed players.");
        bool p1_revealed = players[player_1].revealed;
        bool p2_revealed = players[player_2].revealed;
        if (p1_revealed && !p2_revealed) {
            require(block.number > grace_period + last_commit_block, "Cannot end game until the grace period has passed.");
            players_balances[player_1] += 5 * 1e18;
            players_balances[owner] += 1 * 1e18;
        }
        else if(!p1_revealed && p2_revealed) {
            require(block.number > grace_period + last_commit_block, "Cannot end game until the grace period has passed.");
            players_balances[player_2] += 5 * 1e18;
            players_balances[owner] += 1 * 1e18;
        }
        else if(!p1_revealed && !p2_revealed) {
            require(block.number > grace_period + last_commit_block, "Cannot end game, grace period has not passed yet.");
            players_balances[player_1] += 5 * 1e18;
            players_balances[player_2] += 5 * 1e18;
            players_balances[owner] += 2 * 1e18;
        }
        else {
            require(block.number > grace_period + last_reveal_block, "Cannot end game until the grace period has passed.");
            players_balances[player_1] += 5 * 1e18;
            players_balances[player_2] += 5 * 1e18;
            players_balances[owner] += 2 * 1e18;
        }

        delete players[player_1];
        delete players[player_2];
        delete player_1;
        delete player_2;
    }

    receive() external payable {
        revert();
    }

    fallback() external payable {
        revert();
    }
}
```

Figure 4: Withdraw_balance and forcefully_end_game

## 5 Transaction History

The contract was deployed at the following address:
0xE68Fd5251c301c6E4F37A55061eD80D638C5DED5

My account address:
0x2fda762f105934037f5Aa67827d8733BFa6ae4AE

Transaction history of a game of Morra using my contract (in order of execution):

Creation (tx id):
0x56faa01073f783d73f575ec2ffd37d7028e0f2004fec9b3e49705b73c6fa5d23

My commit call (+6 ETH transfer to the contract):
tx id: 0x86c08586adf148190c7fa1397a238fe8e97bac216520efc837be96805c26ab6d
commit_hash: 0xba0e72cc35d088dce28f051690e6c9df5482bd55da2a9023675223bb372aac3a

Partner's commit call (I only know that she didn't guess my number and that I didn't guess hers, she didn't communicate her safety_number, so I cannot reconstruct the hash) (+6 ETH transfer to the contract):
tx_id: 0xa8f29ee9c8101ac27f8f73cacae351dc48b5f1435946550eb043139a313faad8

My reveal call:
tx_id: 0x1e0dc56e75248d3fe1b80e59b7d262109af1cc295eb623a0ef630e0a9b9a113c
played_nr_and_guess: 9
safety_number: 150

Partner's reveal call (tx id):
0x1f84456284e5fb55ff83e32dcb30c1e195bf0da5af23cac001205c13baa477a4

My call to decide_winner (tx id):
0x8bd9c0346d3eac2c1c25c8163b66855b1e03b019c773314fa0fd560ea9ef1f66

My withdraw_balance call (tx id):
0x6f4926c7440ea7d8d670ce0fe9b14a58d0d8e19ce736d966d24992ad3ac249a2

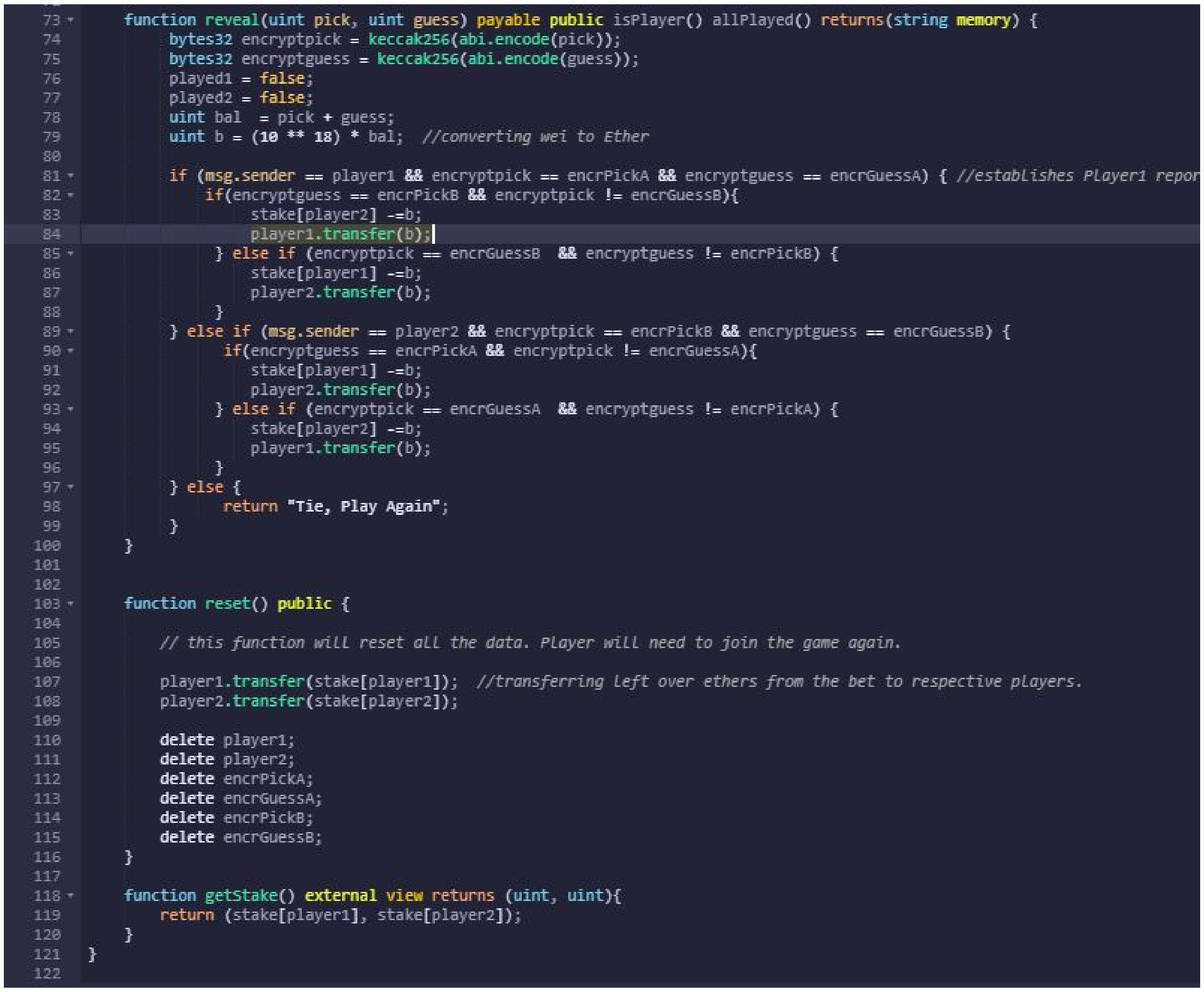
Partner's withdraw_balance call (tx id):
0x5c8b9d563f884fb7c28cd6c2a845b867da63e3f8f95ead33f11a9dfcf608f687

Since the game ended in a draw, we both withdrew the 6 ETH we deposited.

## 6 Commentary on partner's contract

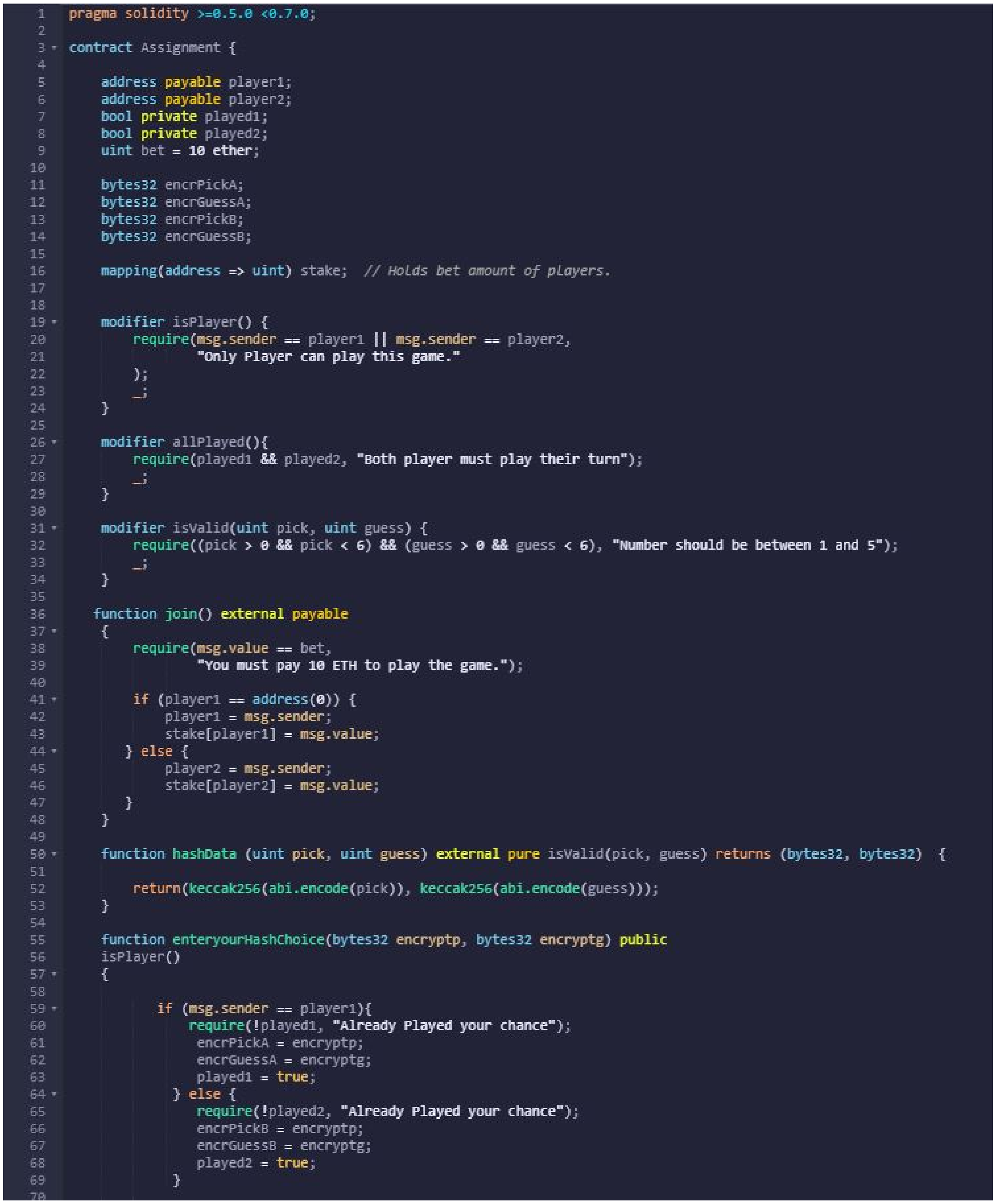
My partner's student number: s2121800.

### 6.1 Partner's contract code



```
    function reveal(uint pick, uint guess) payable public isPlayer() allPlayed() returns(string memory) {
        bytes32 encryptpick = keccak256(abi.encode(pick));
        bytes32 encryptguess = keccak256(abi.encode(guess));
        played1 = false;
        played2 = false;
        uint bal  = pick + guess;
        uint b = (10 ** 18) * bal;  //converting wei to Ether

        if (msg.sender == player1 && encryptpick == encrPickA && encryptguess == encrGuessA) { //establishes Player1 repor
            if(encryptguess == encrPickB && encryptpick != encrGuessB){
                stake[player2] -=b;
                player1.transfer(b);
            } else if (encryptpick == encrGuessB  && encryptguess != encrPickB) {
                stake[player1] -=b;
                player2.transfer(b);
            }
        } else if (msg.sender == player2 && encryptpick == encrPickB && encryptguess == encrGuessB) {
             if(encryptguess == encrPickA && encryptpick != encrGuessA){
                stake[player1] -=b;
                player2.transfer(b);
            } else if (encryptpick == encrGuessA  && encryptguess != encrPickA) {
                stake[player2] -=b;
                player1.transfer(b);
            }
        } else {
            return "Tie, Play Again";
        }
    }


    function reset() public {

        // this function will reset all the data. Player will need to join the game again.

        player1.transfer(stake[player1]);  //transferring left over ethers from the bet to respective players.
        player2.transfer(stake[player2]);

        delete player1;
        delete player2;
        delete encrPickA;
        delete encrGuessA;
        delete encrPickB;
        delete encrGuessB;
    }

    function getStake() external view returns (uint, uint){
        return (stake[player1], stake[player2]);
    }
}
```

Figure 5: Partner contract 2/2



```
pragma solidity >=0.5.0 <0.7.0;

contract Assignment {

    address payable player1;
    address payable player2;
    bool private played1;
    bool private played2;
    uint bet = 10 ether;

    bytes32 encrPickA;
    bytes32 encrGuessA;
    bytes32 encrPickB;
    bytes32 encrGuessB;

    mapping(address => uint) stake;  // Holds bet amount of players.


    modifier isPlayer() {
        require(msg.sender == player1 || msg.sender == player2,
                "Only Player can play this game."
        );
        _;
    }

    modifier allPlayed(){
        require(played1 && played2, "Both player must play their turn");
        _;
    }

    modifier isValid(uint pick, uint guess) {
        require((pick > 0 && pick < 6) && (guess > 0 && guess < 6), "Number should be between 1 and 5");
        _;
    }

   function join() external payable
    {
        require(msg.value == bet,
                "You must pay 10 ETH to play the game.");

        if (player1 == address(0)) {
            player1 = msg.sender;
            stake[player1] = msg.value;
       } else {
            player2 = msg.sender;
            stake[player2] = msg.value;
       }
    }

    function hashData (uint pick, uint guess) external pure isValid(pick, guess) returns (bytes32, bytes32)  {

        return(keccak256(abi.encode(pick)), keccak256(abi.encode(guess)));
    }

    function enteryourHashChoice(bytes32 encryptp, bytes32 encryptg) public
    isPlayer()
    {

           if (msg.sender == player1){
               require(!played1, "Already Played your chance");
                encrPickA = encryptp;
                encrGuessA = encryptg;
                played1 = true;
             } else {
                require(!played2, "Already Played your chance");
                encrPickB = encryptp;
                encrGuessB = encryptg;
                played2 = true;
             }
```

Figure 6: Partner contract 1/2

## 6.2 General remarks

The intended sequence of steps for a game of Morra is the following:

1. Both players pick a number and a guess.
2. They hash these two numbers using the "hashData" method.
3. Then, each of the two players needs to call the "join" method, with a transfer of 10 ETH.
4. The two players submit the hashes obtained in step 2 using the "enteryourHashChoice" function.
5. Then, only one player calls the reveal function and the winner gets his prize. In the case of a draw, the players can play again.
6. At any point, anyone can cancel the game and delete the player data by calling the "reset()" function.

The contract has a lot of vulnerabilities, as we will see in Section 6.4. Also, the contract does not seem to calculate rewards correctly (see line 78). I will comment on the code under the assumption that this flaw has been fixed.

## 6.3 Gas fairness

The join, hashData and enteryourHashChoice functions seem to be symmetrical for the players, so each of them should pay a similar gas fee for using them. However, only one player needs to call the "reveal" function, so he will end up paying more gas. Similarly, only one player should call the reset() function, creating a further gas imbalance. This issue is not addressed in the code, so the gas usage is not fair for the players.

The gas used by enteryourHashChoice and by the contract deployment can be further reduced by eliminating the "played1" and "played2" (lines 7-8) global storage variables. "Played1" for instance can be replaced by the statement: "encrPickA != 0".

## 6.4 Vulnerabilities and flaws

### 6.4.1 Player 2 loses his stake (griefing)

Once both players have joined the game, a third player who wants to produce a loss for the second player can also join the game and then swiftly call reset(). The join() function does not check whether the address for player2 has been assigned or not and blindly overwrites it with the griefer's address. Since the second player's address is no longer saved in the player2 variable, he has no way to retrieve his stake.

This vulnerability can be mitigated with a pull over push scheme: "stake" already memorises each player's balance. All that would be left to implement is a function similar to the withdraw_balance function in my implementation.

### 6.4.2 Incorrect commit-reveal scheme

Let's assume that one of the two players (the "attacker") can somehow intercept the call to "enteryourHashChoice" performed by the other player and get his "encryptp" and "encryptg" numbers. If the other player is an honest player and only picks and guesses numbers between 1 and 5, then the attacker can trivially obtain the honest player's pick and guess from the corresponding hashes (as there are only 5 possible hashes, one for each number) and win the game.

Computing the hashes in a similar manner as in my implementation would solve this issue.

### 6.4.3 Making the game cost more gas for the next two players (griefing)

Two players which are collaborating (or one player with good timing) can increase the gas cost of the game for the next two players. The reason for this is that the reset() function does not set "played1" and "played2" to false. If both players join and enter their hashes normally, both of those variables would be set to true. Then, one of them resets the game. These two booleans remain set to true. The next two players will not be able to enter their hashes (since enteryourHashChoice would not allow them to due to the two bools) without one of them calling the reveal function first (which would result in a draw) in order to reset the two booleans to false.

### 6.4.4 A guaranteed win exploit versus an honest player

Since the reveal function does not check whether the players' number and guess are within 1 and 5, a dishonest player can play a number outside of this range. An honest player will always pick a number in this range. All the dishonest player needs to do is keep playing the game until he guesses the honest player's number (which will happen eventually). Since an honest player will also keep his guesses between 1 and 5, it will be impossible for him to guess the dishonest player's number, so he is guaranteed to lose.

### 6.4.5 Integer underflow

Since the game does not end once a player wins, the players can keep playing after a reveal happens. Since the code does not check whether "stake[player] ¿ b", an integer underflow can occur in any of the following lines: 83, 86, 91, 94. This underflow will prevent the game from ever being ended, since the reset() function will always fail, so the game state cannot be reset.

This underflow (or rather the lack of checking whether the operation will cause an underflow) gives an attacker the chance of stealing all of the funds of another player, providing that the attacker joined the contract before the victim. Using the exploit in 6.4.1, the attacker can use a second account to overwrite the address of the honest player and play with himself in the following way: one of the accounts will always play the number 1, and the other account will keep guessing it, while playing 1 itself. This way, one of the accounts will win 2 ether per game, until the contract balance is depleted. Once the underflow occurs, the game will be left in the same state described in the previous paragraph.

### 6.4.6 Griefing the players

Anyone can call the reset function, ending the game prematurely. The function should be restricted only to the two current players.