BDL Coursework

Mateusz Parafinski
s1554741

November 18, 2018

1. The King of Ether

The highest level overview of the smart contract for the King of Ether is that anyone can become the King by paying sufficient amount of Ether (at least as much as the last King).

When the contract is created, we are saving the information about the address of the owner in the `owner` variable and then automatically the creator becomes the King with the message `Let's play a game...` and with the value 1 wei.

Then we have multiple functions that allow other parties to take part in the game and become the King themselves. `claimThrone(string message)` function allows the user to pay a certain amount of Ether (`payable` keyword) and if the value is greater or equal from the highest value (value that the last King payed), then the user becomes the King with the message they passed to the function. There is also a restriction on the amount of Ether one can use to become the King (50 Ether) that can be lifted by the owner of the contract as described later. Moreover, any time a new King is determined, the last King's earnings are saved in the `earnings` map under his address. This way any user that was the King but got dethroned can recover their money using the `withdraw()` function.

We also have one getter in the form of `getKingsTotal()` that allows to see how many kings were there in total over the lifespan of the contract and `raiseRestriction()` that allows the owner to lift the restriction described two paragraphs above and allow users to pay more than 50 Ether to become the King.

Thankfully I managed to become the King at one point too. The ID of the transaction was

`0x903d0a4f95656cbfdbd520898fdfe811d39119e0d229a4effeb14a93d353a79f`,

my address is

`0x47ADEE763A7BDE2a03c029725C5f7c9315f3B42a`

and the message I used for the transaction was "`Test transaction please ignore`".

2. Rock-paper-scissors

2.1 High level overview

The most general idea behind my implementation is that both players have to pay the same amount of Ether (say $2s$) and then after the game is ended one of them can win at most s from the other. This design decision was made to incentivise both players to finish the game even in case of a loss.

The game starts by a user calling the `play` function with a hashed value of of their choice along with a nonce (for more information about the commitment scheme refer to the section 2.3). Then another person can join the game by calling the `play` function again with the same amount of Ether as the first player and their own hashed nonce-choice pair.

After both players have joined and made their commitments, the game moves to the so-called 'reveal phase', where both players (in any order) have to, as the name suggests, reveal their choices by calling the `reveal` function and providing their choice and the nonce. After both players have done so, the game moves on to the final 'claim prize phase'.

In the last phase the players can, in any order, claim their Ether back. Notice that, as mentioned a few paragraphs above, *both* players have to actually call the `claim` function because they both always get some money back - following from our earlier example with $2s$, in case of a win the player finishes the game with $3s$, in case of a tie with $2s$ and in case of a lose with $1s$. After both players have claimed their Ether the game is reset and a new round can be started.

2.2 Game termination

The scenario described above refers to the expected game flow, however it may be the case that either one or both of the players stop the game mid-through. For this eventuality, there is a timer that will allow a player to reset the game to the start state if any of the players is inactive for a set amount of time (2 minututes).

The way the timer works is that if during the game any of the players is inactive for more than 2 minutes, then any player is able to reset the game to the state where a new game can be started. If both players fail to reveal their values, then any user on the blockchain can call the `claim` function that will send the stakes back to both players from the last game and allow a new game to be initiated. On the other hand, if just one of the players reveals their value and the second one doesn't, the person who doesn't reveal their value gets penalised by not getting any of the Ether back.

It is also worth pointing out that in the current version of the game, once the game is finished and the result is determined, both players are expected to claim back their Ether, because otherwise a new game won't be able to start. This is made with the assumption that players are reasonable and will always want their Ether back.

2.3 Commitment scheme details

The commitment scheme used in the implementation is written with the intention of maximum security and equal gas costs for both players during a round. The protocol starts by both players picking a random nonce and a choice (rock/paper/scissors as either 1, 2 or 3 respectively), hashing both values using Solidity SHA3 function and then sending the hashes to the smart contract. Once both players have done that, they reveal what value they commited to by providing both the nonce and the choice they made. Notice that this works due to the one-way and collision resistance of hash functions - the player cannot reverse hash function to obtain the other players choice but also none of the players is able to change their choice because that would require finding a different nonce that would produce the exact same hash.

Moreover, this seems like a good choice for this protocol due to the fact that both players have to perform the same exact operations which makes the gas costs for both players similar in the first two phases of the protocol ('start' and 'reveal' phases).

In terms of nonce generation, a player could technically come up with a nonce themself, however that may be ill-advised due to low ability to create actual randomness by people. Moreover, asking players to create the nonces by themselves would most likely result in short nonces that would be easy to find. Therefore along with the contract code, I have created a short Python script that, given user's choice (1, 2 or 3) generates a nonce (by taking a random value between 2^{128} and 2^{256}) and then hashes the nonce along with the user's choice. One could argue that Python's `randint` function is not safe enough, although it creates enough randomness to ensure that none of the players will be able to crack the other player's nonce in the time period that the game is on.

2.4 Gas cost analysis

Gas is an intrinsic part of the Ethereum system that allows for the existence of smart contracts. In contracts that consider only a case where one user will interact with the contract we usually only care to make the gas costs as small as possible. However in the case where the contract is fundamentally implemented to allow interaction between two parties, we not only have to care about making the gas costs as low as possible, but also to make them as *equal* as possible. Unfortunately due to the nature of gas it is physically impossible to make the contract use the same exact amount of gas for both parties. We can however try to make the difference as small as possible.

As mentioned in the 2.1 section, the idea behind the game was to make it as symmetric as possible so that both players have to make almost exactly the same steps during the execution of one round of rock-paper-scissors game. With the current implementation of the contract, the player that initiates the game has to pay around 13% more gas than the player joining second over the course of one round of the game.

The difference in gas costs is not extremely big, however such an implementation is obviously far from perfect. First off, if parties know that the player initiating the game will have to pay more gas in order to finish the game, noone may want to be the person starting the game, so we could potentially consider a slight change to the protocol that would result in a lower cost for the *first* player instead of the second to actually incentivise players to start games. Secondly, since the implementation tries to make gas costs similar for both players, there is potentially a lot of space in the code for optimisation that could be done (and would result in less overall gas costs), however that could mean bigger discrepancy in the gas costs between the players. There is definitely a trade-off here that can be best resolved by trial and error with different approaches.

2.5 Attempts at improving the gas costs

Gas system in Solidity is surprisingly (but understandably) complex. Every ‘machine code’ operation has its associated gas cost that the party calling the function has to pay when said operation is executed. While analysing the intricacies of Solidity gas costs, I have come up with a few ideas of how the code could possibly be adjusted in order to equalise/minimise gas usage.

2.5.1 Storage and non-zero states

First very interesting thing about gas costs is the fact that changing a value from a so-called ‘zero state’ to a ‘non-zero state’ results in a SSTORE operation that costs 20000 gas [**?**]. In my contract this resulted the first game after deployment being significantly (∼25%) more expensive than any subsequent game. However, a few small tweaks (such as changing from `true` and `false` to `1` and `2` in the constructor) resulted in a code that had no ‘zero states’ at the beginning of the contract and consequently removed all the SSTORE operations. This seems quite unintuitive, but as far as my understanding goes, working with non-zero integers is more gas efficient than working with boolean values in Solidity.

Another interesting fact about the gas costs is that even if an operation doesn’t change from a ‘zero state’ of a variable to a ‘non-zero state’, it still costs 5000 gas [**?**]. Therefore one idea for reducing the amount of gas required for a game would be to get rid of some not necessarily required global variables in the code. For example, we could potentially get rid of the `has_revealed` variable of each player and try to work with just their `choice` - set choice to a value not in $\{1, 2, 3\}$ when they haven’t yet revealed their choice. This way we would save up on all the transactions that include changing the state of the `has_revealed` variable currently.

One more solution that may result in lower gas costs is shortening the game. In the current implementation both players have to call the contract three times during the execution of a single game. We could, at the cost of gas assymmetry make the game shorter and include the Ether claiming phase of the game in the reveal phase and use `claim` only if one of the players doesn’t finish the game in time. A bit of testing shows that this approach indeed results in smaller gas costs in comparison to the original method (∼200000 for player 1 and ∼150000 for player 2), however the gas costs differ by around 25% between the players.

The lesson here should be that storage in Solidity is considered very expensive and the more 'stateless' the contract the cheaper it will be.

2.5.2 Asymmetric protocol

Another approach that I have considered for the contract is an asymmetric protocol where the first player starts the game, the second one joins and then the one who reveals second gets their money during the reveal call, whereas the other player has to call the `claim` method in order to retrieve the ether. Surprisingly, even though the asymmetric nature could lead to an assumption that gas costs cannot be similar, this protocol seems to perform best in this regard. Both players pay roughly 220000 gas for the whole game which is at the same level as the original game but with around 5% difference between the gas costs of each player. This seems like a good approach although again, thorough testing should be done in order to tell with hundred percent certainty that this is the way to go.

2.6 Other people's vulnerabilities

- Addition of nonce to choice
- Timestamp attack of a miner
- Can always reset before reveal (Wes)
- Reentrancy in my old code
- Didn't check who revealed (Me)
- Didn't check who got paid already (Me)
- Activity attack (wait for inactive, Me)

3. Game code

Below you can find the game code

```
pragma solidity ^0.4.16;

contract rpsContract {

    // This should be self-explanatory
    struct Player {
        address add;
        bool revealed;
        bool got_paid;
        uint256 hashed_choice;
        uint8 choice;
    }

    address owner;
    // Count if both players got paid (in terms of a tie)
    Player[2] players;
    // 0 - player 0, 1 - player 1, 2 - tie
    uint gameWinner;
    // Used to reset the game in case of inactivity
    uint256 timer;
    GamePhase gamePhase;
    uint256 gameStake;

    // Idle - waiting for new players
    // Started - one player started the game
    // Reveal - waiting for the players to reveal
    // Finished - ready to claim the Ether
    enum GamePhase { Idle, Started, Reveal, Finished }

    constructor () public {
        owner = msg.sender;
        gamePhase = GamePhase.Idle;
        gameStake = 0;
        players[0] = Player(0, false, false, 0, 0);
        players[1] = Player(0, false, false, 0, 0);
    }

    /**
    Start a new game or get into an existing one by sending a hashed (sha3)
    value of the choice and a random (chosen by the player) seed
     */
    function play(uint256 hashed_choice) public payable {
        require (gamePhase == GamePhase.Idle || gamePhase == GamePhase.Started, "Game is currently on");
        require (msg.value >= 1 ether, "Stakes need to be at least 1 ether");
        require (msg.value % 2 == 0, "Stakes need to be divisible by 2");

        if (gamePhase == GamePhase.Idle) { // first player starting the game
            // half of what you put in is treated as deposit
            // and half as the game prize pool
            gameStake = msg.value / 2;
            gamePhase = GamePhase.Started;
            players[0] = Player(msg.sender, false, false, hashed_choice, 0);
        } else { // second player joining the game
            require (msg.value / 2 == gameStake, "Stake needs to be equal to the other player's stake");
            gamePhase = GamePhase.Reveal;
            // start the timer to enable resetting the game
            // in case of inactivity
            timer = now;
            players[1] = Player(msg.sender, false, false, hashed_choice, 0);
        }
    }

    /**
    Reveal your choice by providing the nonce and choice
     */
    function reveal(uint256 nonce, uint8 choice) public {
        require(gamePhase == GamePhase.Reveal, "Game not in reveal phase yet");
        bytes memory reveal_val = abi.encodePacked(nonce, choice);
        uint8 p = determinePlayer(msg.sender);
        require(p != 2, "You are not taking part in the game");
        require(uint256(keccak256(reveal_val)) == players[p].hashed_choice, "Invalid seed and/or choice");

        // save the choice and mark that
        // the player have revealed the value
        players[p].choice = choice;
        players[p].revealed = true;

        if (players[0].revealed == true && players[1].revealed == true) {
            // both players revealed
            // determine winner and allow claiming winnings
            gamePhase = GamePhase.Finished;
            gameWinner = getWinner(int8(players[0].choice), int8(players[1].choice));
        } else {
            // reset the timer due to activity
            timer = now;
        }
    }

    /**
    Claim money or reset the game in case of inactivity
     */
    function claim() public {
        if (now - timer > 2 minutes && gamePhase == GamePhase.Reveal) {
            // prevent reentrancy attack
            reset();

            // 2 minutes have passed since last activity
            // allow anyone to reset the game
            if (players[0].revealed == true) {
                // first player revealed, gets all
                players[0].got_paid = true;
                players[0].add.transfer(4*gameStake);
            } else if (players[1].revealed == true) {
                // second player revealed, gets all
                players[1].got_paid = true;
                players[1].add.transfer(4*gameStake);
            } else {
                // no player revealed, split 50/50
                players[0].got_paid = true;
                players[1].got_paid = true;
                players[0].add.transfer(2*gameStake);
                players[1].add.transfer(2*gameStake);
            }

            return;
        }

        require(gamePhase == GamePhase.Finished, "Game not finished yet");
        uint8 p = determinePlayer(msg.sender);
        require(p != 2, "You are not taking part in the game");
        require(!players[p].got_paid, "You already have your money");

        if (gameWinner == p) {
            players[p].got_paid = true;
            // transfer 75% of total to winner
            msg.sender.transfer(3*gameStake);
        } else if (gameWinner != p && gameWinner != 2) {
            players[p].got_paid = true;
            // transfer 25% of total to loser
            msg.sender.transfer(gameStake);
        } else if (gameWinner == 2) {
            players[p].got_paid = true;
            // transfer each player 50% of total
            msg.sender.transfer(2*gameStake);
        }

        if (players[0].got_paid && players[1].got_paid) {
            reset();
        }
    }

    /**
    Determine the player based on the address.
    Returns 2 if player is unknown
     */
    function determinePlayer(address add) private view returns(uint8) {
        if (players[0].add == add) {
            return 0;
        } else if (players[1].add == add) {
            return 1;
        } else {
            return 2;
        }
    }

    /**
    Reset the game state to idle to allow new games
    to be played
     */
    function reset() private {
        gamePhase = GamePhase.Idle;
    }

    /**
    Determine the winner given choices of both players
    0 + 3k means 'rock'
    1 + 3k means 'paper'
    2 + 3k means 'scissors'
     */
    function getWinner(int8 a, int8 b) public view returns (uint8) {
        require(gamePhase == GamePhase.Finished, "Game not finished yet");

        if ((a - b) % 3 == 1) {
            return 0;
        } else if ((a - b) % 3 == -1) {
            return 1;
        } else if ((a - b) % 3 == 2) {
            return 1;
        } else if ((a - b) % 3 == -2) {
            return 0;
        } else if ((a - b) % 3 == 0) {
            return 2;
        }
    }

    /**
    Helper function to be able to see what state
    the game is currently in
     */
    function getGamePhase() public view returns (string) {
        if (gamePhase == GamePhase.Idle) {
            return "Waiting for players";
        } else if (gamePhase == GamePhase.Started) {
            return "Waiting for player 2";
        } else if (gamePhase == GamePhase.Reveal) {
            return "Waiting for the players to reveal their choices";
        } else if (gamePhase == GamePhase.Finished) {
            if (gameWinner == 0) {
                return "Player 1 won, waiting to claim the prize";
            } else if (gameWinner == 1) {
                return "Player 2 won, waiting to claim the prize";
            } else {
                return "Tie, waiting for the players to claim the money";
            }
        }
    }

    /**
    Helper function to be able to see what is the
    stake you need to put in to compete
     */
    function getStake() public view returns (uint256) {
        return gameStake;
    }
}
```

and the hash generating script in Python

```
from web3 import Web3
from random import randint

seed = randint(2**128, 2**256)
choice = int(input("Pick a choice: "))

h = Web3.soliditySha3(['uint256', 'uint8'], [seed, choice%3])
print("Seed-choice:", str(seed) + ", " + str(choice%3))
print("Hash:", int(h.hex(), 16))
```