BDL Coursework

Mateusz Parafinski
s1554741

November 20, 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 with the bonus equal to the difference between their value and the new King's value 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 x) and then after the game is ended one of them can win at most $\frac{x}{2}$ 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 stake x, in case of a win the player finishes the game with $\frac{3x}{2}$, in case of a tie with x and in case of a lose with $\frac{x}{2}$. 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 ($\sim$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 Vulnerability analysis

When writing a smart contract, many things can go wrong or unnoticed and then be used maliciously by other parties. I am going to go through some of the vulnerabilities I have found when analysing other people's code over the past few weeks.

2.6.1 Reentrancy

One of the very common vulnerabilities of smart contracts is the ability to call back a function after being transferred some funds before the state of the function is updated to get more funds out of the contract. Even though most students have taken into account the fact that reentrancy is dangerous, it is not difficult to forget about some possible cases where this may happen. In one example, a student made sure to reset the state if the game ended in the predicted way by revealing the values by both players. However, in the case where one or both of the players were inactive and sort of a reset had to be called, the state update was called after transfering the money to the players. This obviously leads to the possibility of a reentrancy attack. In code, this looks something along those lines:

```
/**
...
*/
function claimMoney() {
    if (msg.sender == winner) {
        // here we prevent the reentrancy
        updateState();
        msg.sender.transfer(winnings);
    }
}

function kickInactive() {
    if (now - time > 1 hour) {
        player1.transfer(stake/2);
        player2.transfer(stake/2);
        // here reentrancy is possible
        updateState();
    }
}
/**
...
*/
```

Obviously `player1` can exploit the vulnerability by simply running the `kickInactive` function again when the money is being transferred to them and get all the money instead of just half. The lesson here is to make sure every time we are either transfering money or calling a function of a different contract that the state of our contract doesn't allow any unwanted actions to be taken.

2.6.2 Checking identities

Another relatively simple vulnerability that I have noticed is that whenever one of the players gets paid, it isn't actually checked **who** gets paid and thus the same player can call the function in quick succession and get all the money out of the contract. An example of this attack can be seen here

```
/**
...
*/
function claimWinnings() {
    if (msg.sender == winner) {
        msg.sender.transfer(winnings + deposit);
    } else {
        msg.sender.transfer(deposit);
    }
}
/**
...
*/
```

Obviously, this attack only works in a symmetric scheme in which both players have to call the `claimWinnings` function to claim their money back, i.e. if there is some kind of deposit that both of them put in at the start of the game. If only the winner gets money, then they will get all the money out of the contract in the first run of the function anyway. Nevertheless, if a contract involves interaction between two parties and the way the contract should behave differs by player, then we should always check for the identity of the player and update the state based on the action we have done as in here (based on the code from Section 3):

```
/**
...
*/
function claimWinnings() {
    player = determinePlayer(msg.sender);
    require(!player.gotPaid, "Player already got paid");
    // here we make sure the state gets updated
    // before the function can be called again
    player.gotPaid = true;

    if (player == winner) {
        player.transfer(winnings + deposit);
    } else {
        player.transfer(deposit);
    }
}
/**
...
*/
```

2.6.3 'Inactivity' attack

This is a bit more subtle vulnerability that makes use of the fact that someone has to start the game and wait for an opponent. Let's say that happens and we are waiting for player 2 to join the game. What player 2 could do is wait until player 1 forgets that they have started the game (or simply is bored of waiting longer for an opponent) and then start the game and immediately reveal their hand.

Now, we know that to prevent a player from never revealing their hand, we need to include a timer in the implementation to be able to reset in case of inactivity. Therefore, since player 1 has forgot about the game/is inactive for some other reason as planned by malicious player 2, player 2 can just wait and claim all the money.

Since most games I have analysed had the same model of the game flow, this attack works for most of them and is not really a flaw if we implement some counter-measures that player 1 can take in order to prevent this from happening. One such measure would be to enable player 1 to play with themself if noone else decides to play with them, which can be seen in the following code:

```
/**
...
*/
function hashHand(bytes32 hash) payable public {
    require(gameState <= 2);

    // enforce deposit for not completing protocol punishment
    require(msg.value >= minBuyIn + minDeposit);

    if (gameState == 1) {
        gameState = 2;
        player1State = PlayerState(msg.sender, hash, msg.value - minDeposit);
        // the player stakes exclude the mandatory deposit (it does not take
        // part in the prize-pool)

    } else { // this must be gameState = 2
        gameState = 3;
        player2State = PlayerState(msg.sender, hash, msg.value - minDeposit);
        time = now; // start timer for detecting inactivity

        // this is where "residuals", which is the overpaid excess money
        // that should be returned to their respective owners is tracked
        if (player1State.stake >= player2State.stake) {
            winnings = player2State.stake; // also calculating prize-pool
            p1_residual = player1State.stake - player2State.stake + 1;
            // offseting the residuals by 1 to avoid zeroes
        } else {
            winnings = player1State.stake;
            p2_residual = player2State.stake - player1State.stake + 1;
        }
    }
}
/**
...
*/
function revealHand(uint256 hand) public {
    require(gameState >= 3);
    if (gameState == 3) {

        require(msg.sender == player2State.adress);
        require(sha256(abi.encodePacked(hand)) == player2State.hash);
        // the hash-based commitment scheme check
        gameState = 4;

        p2_hand = hand % 10 + 3; // saving with the offset for avoiding zeroes
        time = now; // overwrite timer for detecting inactivity

    } else { // this must be gameState = 4

        require(msg.sender == player1State.adress);
        require(sha256(abi.encodePacked(hand)) == player1State.hash);

        hand = hand % 10;
        bool isDraw = (hand % 3 == p2_hand % 3);
        bool player2Won = (hand % 3 == (p2_hand + 1) % 3);

        if (isDraw){
            endGameGetPaid(player1State.adress, player2State.adress,
                            player1State.stake, player2State.stake, 0, false);
                            // indicate draw with a 0-value winning,
                            // residuals become the original stakes
        } else {
            if (player2Won) {
                endGameGetPaid(player2State.adress, player1State.adress,
                                p2_residual, p1_residual, winnings, false);
            } else {
                endGameGetPaid(player1State.adress, player2State.adress,
                                p1_residual, p2_residual, winnings, false);
            }
        }
    }
}
/**
...
*/
```

We can see that there is nothing here preventing player 1 to, in case of noone else wanting to play, start a game with themself and play it out and get all the money back no matter of the result. Another possible approach to solving this issue (with possibly smaller gas costs) would be to allow a player to reset the game state and get the money back as long as there is just one player in the game, e.g. something like

```
/**
...
*/
function withdraw() {
    require(gamePhase == WaitingForPlayer2);
    gamePhase = WaitingForPlayers;
    player1.transfer(stake);
}
/**
...
*/
```

2.6.4 Hash calculation in the commitment

In all the protocols I have looked at (including mine), the second phase of the game is related to revealing the commited value by both players. Usually, the players provide the nonce and the choice as the arguments to the revealing function which then checks whether the values they have given hash to their earlier commitment.

There are multiple ways that a protocol could define ‘combining’ the values of the nonce and the choice. I would like to focus here on two that I have found to be vulnerable to attacks. From this point on I will denote the nonce by n and the choice by c.

The first and simpler version combines the values of nonce and choice by simply adding them together: $n+c$. One contract that I have looked at had this exact procedure for checking the commitment:

```
/**
...
*/
function reveal(uint256 randInt, RPS choice) private {
    require(registered(), "You have not registered for the game!");
    require(gameInProgress(), "At least one player missing!");

    // player1 always reveals his choice and random integer first.
    if (gameState == State.NoReveal1 && msg.sender == player1) {
        // Make sure that the hashes match and thus player1's commitement was valid.

        // *** BELOW IS THE LINE WITH THE VULNERABILITY ***
        require(player1Hash == sha256(bytes32(randInt + uint256(choice))), "Hashes 1 don't
            match!");

        // Store the random integer and choice of player1.
        player1randInt = randInt;
        player1Choice = choice;
        // Update the game sate to make sure it's player2's turn to reveal next.
        gameState = State.NoReveal2;

        // Update the time stamp to check for timeout later
        timerStart = now;
    }
    /**
    ...
    */
}
/**
...
*/
```

As we can see, in the code, the contract creator decided to combine the nonce and the choice by adding them together. However, this leads to a quite devastating attack - imagine the player initially chose nonce n and choice c and got a hash $h = \text{sha256}(n + c)$. Notice that any pair $(n - k, c + k)$ will lead to the same hash:

$$\text{sha256}(n - k + c + k) = \text{sha256}(n + c) = h$$

which means that essentially a player can change their choice at any given point before revealing - in particular, they can change it after the other person reveals their choice. Obviously this is far from desired.

After I have notified the author of the vulnerability, they came up with another protocol that instead of adding the values together 'appended' the choice at the end of the nonce, so basically the joined value was calculated as

$$10n + (c\%3)$$

and then hashed to produce

$$h = \text{sha256}(10n + (c\%3)).$$

Is this perfectly safe? Certainly the attack suggested earlier will not work - providing $n - k$ as nonce and $c + k$ as choice leads to $h' = \text{sha256}(10n - 10k + ((c + k)\%3)))$ which with very high probability leads to a hash different than our original one, $h \neq h'$. However, is there by any chance another attack that could work here? Consider the following corrected version of the code from above:

```
function revealRock(uint256 randInt) public{
    // 'revealRock' reveals that the sender choice was Rock, and randInt in the random
        integer he
    // used to generate the hash.
    reveal(randInt, RPS.Rock); // RPS.Rock is 0
}

function revealPaper(uint256 randInt) public{
    // 'revealPaper' reveals that the sender choice was Paper, and randInt in the random
        integer he
    // used to generate the hash.
    reveal(randInt, RPS.Paper); // RPS.Paper is 1
}

function revealScissors(uint256 randInt) public{
    // 'revealScissors' reveals that the sender choice was Scissors, and randInt in the
        random integer he
    // used to generate the hash.
    reveal(randInt, RPS.Scissors); // RPS.Scissors is 2
}

function reveal(uint256 randInt, RPS choice) private {
    require(registered(), "You have not registered for the game!");
    require(gameInProgress(), "At least one player missing!");

    // player1 always reveals his choice and random integer first.
    if (gameState == State.NoReveal1 && msg.sender == player1) {
        // Make sure that the hashes match and thus player1's commitement was valid.
        require(player1Hash == sha256(bytes32(randInt*10 + uint256(choice))), "Hashes 1 don'
            t match!");
        // Store the random integer and choice of player1.
        player1randInt = randInt;
        player1Choice = choice;
        // Update the game sate to make sure it's player2's turn to reveal next.
        gameState = State.NoReveal2;

        // Update the time stamp to check for timeout later
        timerStart = now;
    }
    /**
    ...
    */
}
```

I claim that this implementation leads to a very effective attack that guarantees $\frac{2}{3}$ chance of a win and $\frac{1}{3}$ chance of a draw against an opponent that chooses each of the three possibilities uniformly. The key to the attack is the fact that Solidity, similar to many other programming languages, works with numbers using modular arithmetic. That is, if we have a variable of type `uint8` with value 250, say `uint8 v = 250`, then `v + 10` will not result in `v` being equal to 260 but to $260\%256 = 4$. The same goes for `uint256` but the modular arithmetic is done modulo 2^{256}.

The idea is that if we manage to overflow the value of `randInt*10` in the code above, then maybe we are able to find a 'collision' that allows us to change our choice. Thus we are focused in finding two numbers, say a and b such that

$$10a + k = 10b + l \pmod{2^{256}}$$

where k and l are two choices that we could potentially make. We can rewrite the equation above to produce

$$10(a - b) = l - k \pmod{2^{256}}$$

so if we set $x := a - b$, then we are looking for an x and m such that

$$10x + m2^{256} = l - k.$$

It's not hard to prove that this has a solution if and only if $\gcd(10, 2^{256}) | l - k$ and therefore since $\gcd(10, 2^{256}) = 2$ we need l and k to be such that $2 | l - k$. Now, since the code above only allows $l, k \in \{0, 1, 2\}$, this will only work for $l = 2$ and $k = 0$ or vice-versa, so we can only change between Rock and Scissors. How do we calculate x and m? Here the Extended Euclidean Algorithm helps us - this simple code below allows us to calculate the values we are looking for in a matter of milliseconds:

```python
def xgcd(b, a):
    x0, x1, y0, y1 = 1, 0, 0, 1
    while a != 0:
        q, b, a = b // a, a, b % a
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1
    return  b, x0, y0
```

Running this as `xgcd(10, 2**256)` will return three values - the greatest common divisor (2) and x and m we have been looking for. And here, we want to focus on x, which is the 'magic number' that allows us to perform the attack[1]. Following the reasoning from above, we can now pick any two numbers a and b such that $a - b = x$ and guarantee that $10a = 10b + 2 \pmod{2^{256}}$, which in turn allows us to always start the game by picking rock and nonce a and then, if we see that the opponent has revealed paper, change it to scissors and nonce b. This way we win any time the opponent picks paper or scissors and draw whenever they pick rock, giving the desired 66.(6)% chance of a win and 33.(3)% change of a draw.

The script below produces a random pair (a, b) described above to facilitate the attack:

```python
import hashlib, random

magic = 92633671389852956338856788006950326282615987732512451231566067206330503711949

def attack():
    rand_int = random.randrange(0, magic)
    rock = magic + rand_int
    print("rock: {}".format(rock))
    rock = (rock * 10) % 2**256
    rock = rock.to_bytes(32, byteorder='big')
    rock_hash = hashlib.sha256(rock).hexdigest()
```

[1] $x = 92633671389852956338856788006950326282615987732512451231566067206330503711949$

```
    scissors = rand_int
    print("scissors: {}".format(scissors))
    scissors = ((scissors * 10) + 2) % 2**256
    scissors = scissors.to_bytes(32, byteorder='big')
    scissors_hash = hashlib.sha256(scissors).hexdigest()

    print("rock hash: 0x{}".format(str(rock_hash)))
    print("scis hash: 0x{}".format(str(scissors_hash)))

attack()
```

2.6.5 Timestamp attack

Another very subtle type of attack requires one of the players to be a miner or to cooperate with one. As we know, in Ethereum the block timestamp tolerance is 900 seconds (15 minutes) [1]. Therefore, a miner can post ones transaction on the block as if it happened 15 minutes later than the current time.

Imagine the following situation. A miner is one of the players in the game and has the ability to reveal their hand first (this is an important part that will allow us to perform the attack). What the miner can do is reveal their value after the other player joins the game and then *immediately* call the `finish` function in the code below and hope to mine the block with his call to the `finish` function. If he manages to mine the block, he sets the block's timestamp to `now + k` for $k > 100$ and thus the function will be executed as if the time limit has passed and the other player did not reveal their choice, so the miner will get all the money for that game.

```
function reveal(string nonce, uint choice) public returns (string){
    /**
    ...
    */

    //record time of first reveal
    if (timeReveal==0) {
        timeReveal = now;
    }

    /**
    ...
    */
}

function finish() public {
    //both players need to have revealed or the second player needs to have missed the
        reveal window
    require ((hasRevealed[player1] && hasRevealed[player2]) || now-timeReveal > 100);

    //avoid re-entrancy
    uint payout = pot;
    pot = 0;

    //if one party did not reveal, pay the other party
    if (hasRevealed[player1] && !hasRevealed[player2]) {
        player1.transfer(payout);
        payout=0;
    }

    if (hasRevealed[player2] && !hasRevealed[player1]) {
        player2.transfer(payout);
        payout=0;
    }

    /**
    ...
    */
}
```

The obvious way to prevent this kind of attack would be to change the time check to a value bigger than 900 seconds (e.g. 20 minutes = 1200 second) so that the miner cannot immediately call the `finish` function and claim all the money as the check `now - timeReveal > 1200` won't pass anymore.

It is however worth noting that two popular Ethereum protocol implementations Geth and Parity both reject blocks with timestamps more than **15 seconds** in the future [2, 3, 4]. Therefore there exists a so-called 15-second rule that states that "If the contract function can tolerate a 15-second drift in time, it is safe to use block.timestamp" [2], so this would most likely be safe anyway.

3. Game code

Below you can find the game code for my implementation of the rock-paper-scissors game.

```
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))
```

References

[1] Ethereum Block Protocol 2.0, wiki post created by Github users **ruchevits** and **heiko-heiko**, https://github.com/ethereum/wiki/blob/c02254611f218f43cbb07517ca8e5d00fd6d6d75/Block-Protocol-2.0.md

[2] Recommendations for Smart Contract Security in Solidity, https://consensys.github.io/smart-contract-best-practices/recommendations/#timestamp-dependence

[3] Geth protocol implementation of the timestamp rule, https://github.com/ethereum/go-ethereum/blob/4e474c74dc2ac1d26b339c32064d0bac98775e77/consensus/ethash/consensus.go#L45

[4] Parity protocol implementation of the timestamp rule, https://github.com/paritytech/parity-ethereum/blob/73db5dda8c0109bb6bc1392624875078f973be14/ethcore/src/verification/verification.rs#L296-L307