

BDL Coursework

Mateusz Parafinski
s1554741

November 23, 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 to 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. 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.

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 (with the rest being the deposit returned to both players). The design decision of having a big deposit 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 their choice along with a nonce (for more information about the commitment scheme refer to Section 2.3). Then another person¹ can join

¹The second player could actually be the same person as the first one for security reasons - see Section 2.7.3.

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 anyone to reset the game to the start state in case of inactivity.

The timer has a threshold of 2 minutes and is updated to current time at every activity after the game starts (i.e. after both players provide a commitment). Now, if the timer hasn’t been updated in 2 minutes, then any blockchain user can interact with the contract by calling the `claim` function. The function will, based on whether any player has revealed their value or not, send either half of the money to both players (so each of them will get their investment back) or everything to just one of them in case of just one player failing to reveal - note that we are penalising inactivity more than actual loss in the game, the player doesn’t get anything back if they bail.

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 and that withholding the funds inside the contract has no use to any party.

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 0, 1 or 2 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 committed to by providing both the nonce and the choice they made. Notice that this protocol works due to the fact that hash functions are one-way and collision resistant - 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) - see Section 2.5.

In terms of nonce generation, a player could technically come up with a nonce themselves, however that may be ill-advised due to the fact that people are bad at making things random. 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 (0, 1 or 2) 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 potentially Python’s `randint` function may not be 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 Transaction history of the game execution

Now that we know how the game works and what steps are needed to play one game of rock-paper-scissors on the contract, we can go through a history of transactions that were made to have a go at it. The game was played on my contract with the address

0x991a0c2c51d9b5772151fbd92569777869cca499.

The game started by me calling the function `play` with my hash as the parameter:

hash 1: 57422681862702588708287488993004352428084682103926404673879216601876390030369

The transaction ID for this call was

0x260d376fe852f28e8ccbd3aaec29fcdc1f50a43bc0bc762d1889d5a7bfdb2af4.

Next, my friend joined the game by also providing their hash to the `play` function:

hash 2: 91798907740737038578965130316324719596937870764908358325218676231228180405586.

The transaction ID for this call was

0x970781723b179228e55c48a26aa2955e8e81226ded0f9a0d211aa2e498e2fec4.

Now, the contract was waiting for us to reveal the values in any order. I was the first one to reveal mine by calling the `reveal` function with the following parameters:

choice 1: 1 (Paper)

nonce 1: 68467566110241134270540132358642702736493862980289395847769434602692193680791.

The transaction ID for this call was

0x75e54a594aa17d23a8bf03e9bd1effe876de571bc2df2a918accc73071b88092.

Subsequently, my friend had to reveal their choice too using the `reveal` function:

choice 2: 1 (Paper)

nonce 2: 65958972353641206791692374757714317624675152672570927745663838423713800289627.

The transaction ID for this call was

0xc959192c57ecb834f31507923ff363361ad7b4f3d91afeb7f06349d04f4a2081.

At this point the game was technically finished and the winner (or actually lack thereof) was determined, but we still had to claim back our money with the help of `claim` function. I was the first one to do that. I claimed back my 1 Ether and the transaction ID for this was

0x83c223afff33453d08ae31aaab2db91f274b8bd0f2a83f150662b5a946d540ce.

The last thing left to do was for my friend to get their money back. He called the `claim` function again and retrieved his 1 Ether. The transaction ID for this call was

0xcfa840a8a7ce927f95db7209fee0d6215f7f0e500b5d00f07d9332008fbc40aa.

At this point the contract is ready for the next two players to join the game as can be checked by calling the `getGamePhase` function.

2.5 Gas cost analysis

Gas is an intrinsic part of the Ethereum system that allows for the existence of smart contracts. In contracts which don't allow interaction between multiple users, 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 Section 2.1, the idea behind the game was to make it symmetric so that both players have to take almost exactly the same steps during the execution of one rock-paper-scissors game. With the current implementation of the contract, the player that initiates the game has to pay around 10% more gas than the player joining second over the course of one the game. Table 1 represents the average gas costs for each order of the play possible:

Player	Start phase [gas]	Reveal phase [gas]	Claim phase [gas]	Average cost [gas]
Player 1 starts, Player 1 reveals 1st, Player 1 claims 1st				
1	50008	54393	36614	141014
2	35460	60417	27096	123270
Player 1 starts, Player 1 reveals 1st, Player 2 claims 1st				
1	50008	54393	26855	131255
2	35467	60714	36570	132751
Player 1 starts, Player 2 reveals 1st, Player 1 claims 1st				
1	50008	60974	36730	147712
2	35431	54703	27299	117433
Player 1 starts, Player 2 reveals 1st, Player 2 claims 1st				
1	50008	60495	26855	137358
2	35438	54397	36570	126405

Table 1: Average gas costs in different configurations. The average was taken by calculating all possible combinations of choices for each player

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 a lot of potential for optimisation in the code (that 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.

2.6 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.

2.6.1 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 [1]. In my contract this resulted in 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 Solidity².

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’ but to a different ‘zero state’, it still costs 5000 gas [1]. 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 $\{0, 1, 2\}$ 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.

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 asymmetry 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 (~ 120000 for player 1 and ~ 100000 for player 2), however gas costs differ by around 20% 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.

2.6.2 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 120000 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 instead of 10%. 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.

2.6.3 Forced order

Finally, one last thing to notice when looking at the Table 1 is that in the case when player 1 reveals first and player 2 claims first the gas costs are nearly identical. Thus another approach at making the costs equal could be dictating a strict order of play where player 2 cannot reveal first and player 1 cannot claim the money first.

2.7 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.

²The gas costs for the `SSTORE` operations were moved to the constructor instead, so the owner has to pay more, nothing is free.

2.7.1 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 smart contract is updated. 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 usual manner. However, in the case where one or both of the players were inactive some kind of a reset had to be called, the state update was called after transferring the money to the players. This obviously leads to the possibility of a reentrancy attack. In the code, this could look like:

```
1  /**
2  ...
3  */
4  function claimMoney() {
5      if (msg.sender == winner) {
6          // here we prevent the reentrancy
7          updateState();
8          msg.sender.transfer(winnings);
9      }
10 }
11
12 function kickInactive() {
13     if (now - time > 1 hour) {
14         player1.transfer(stake/2);
15         player2.transfer(stake/2);
16         // here reentrancy is possible
17         updateState();
18     }
19 }
20 /**
21 ...
22 */
```

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 that every time we are either transferring money or calling a function of a different contract, the state of our contract doesn't allow any unwanted actions to be taken.

2.7.2 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

```
1  /**
2  ...
3  */
4  function claimWinnings() {
5      if (msg.sender == winner) {
6          msg.sender.transfer(winnings + deposit);
7      } else {
8          msg.sender.transfer(deposit);
9      }
10 }
11 /**
12 ...
13 */
```

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):

```

1  /**
2  ...
3  */
4  function claimWinnings() {
5      player = determinePlayer(msg.sender);
6      require(!player.gotPaid, "Player already got paid");
7      // here we make sure the state gets updated
8      // before the function can be called again
9      player.gotPaid = true;
10
11      if (player == winner) {
12          player.transfer(winnings + deposit);
13      } else {
14          player.transfer(deposit);
15      }
16  }
17  /**
18  ...
19  */

```

2.7.3 ‘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 gets bored of waiting 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, 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, but I wouldn’t call it an implementation flaw. We can prevent this from happening by introducing some counter-measures that player 1 can take. One such measure would be to enable player 1 to play with themselves if noone else decides to play with them, which can be seen in the following code:

```

1  /**
2  ...
3  */
4  function hashHand(bytes32 hash) payable public {
5      require(gameState <= 2);
6
7      // enforce deposit for not completing protocol punishment
8      require(msg.value >= minBuyIn + minDeposit);
9
10     if (gameState == 1) {
11         gameState = 2;
12         player1State = PlayerState(msg.sender, hash, msg.value - minDeposit);
13         // the player stakes exclude the mandatory deposit (it does not take
14         // part in the prize-pool)
15
16     } else { // this must be gameState = 2
17         gameState = 3;
18         player2State = PlayerState(msg.sender, hash, msg.value - minDeposit);
19         time = now; // start timer for detecting inactivity
20
21         // this is where "residuals", which is the overpaid excess money
22         // that should be returned to their respective owners is tracked
23         if (player1State.stake >= player2State.stake) {
24             winnings = player2State.stake; // also calculating prize-pool
25             p1_residual = player1State.stake - player2State.stake + 1;
26             // offsetting the residuals by 1 to avoid zeroes
27         } else {
28             winnings = player1State.stake;

```

```

29         p2_residual = player2State.stake - player1State.stake + 1;
30     }
31 }
32 }
33 /**
34 ...
35 */
36 function revealHand(uint256 hand) public {
37     require(gameState >= 3);
38     if (gameState == 3) {
39
40         require(msg.sender == player2State.address);
41         require(sha256(abi.encodePacked(hand)) == player2State.hash);
42         // the hash-based commitment scheme check
43         gameState = 4;
44
45         p2_hand = hand % 10 + 3; // saving with the offset for avoiding zeroes
46         time = now; // overwrite timer for detecting inactivity
47
48     } else { // this must be gameState = 4
49
50         require(msg.sender == player1State.address);
51         require(sha256(abi.encodePacked(hand)) == player1State.hash);
52
53         hand = hand % 10;
54         bool isDraw = (hand % 3 == p2_hand % 3);
55         bool player2Won = (hand % 3 == (p2_hand + 1) % 3);
56
57         if (isDraw){
58             endGameGetPaid(player1State.address, player2State.address,
59                             player1State.stake, player2State.stake, 0, false);
60             // indicate draw with a 0-value winning,
61             // residuals become the original stakes
62         } else {
63             if (player2Won) {
64                 endGameGetPaid(player2State.address, player1State.address,
65                                 p2_residual, p1_residual, winnings, false);
66             } else {
67                 endGameGetPaid(player1State.address, player2State.address,
68                                 p1_residual, p2_residual, winnings, false);
69             }
70         }
71     }
72 }
73 /**
74 ...
75 */

```

We can see that there is nothing here preventing player 1, in case of no one else wanting to play, to start a game with themselves 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

```

1  /**
2  ...
3  */
4  function withdraw() {
5      require(gamePhase == WaitingForPlayer2);
6      gamePhase = WaitingForPlayers;
7      player1.transfer(stake);
8  }
9  /**
10 ...
11 */

```


2.7.4 Hash calculation in the commitment

In all the implementations I have looked at, the second phase of the game is related to revealing the committed 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:

```
1  /**
2  ...
3  */
4  function reveal(uint256 randInt, RPS choice) private {
5      require(registered(), "You have not registered for the game!");
6      require(gameInProgress(), "At least one player missing!");
7
8      // player1 always reveals his choice and random integer first.
9      if (gameState == State.NoReveal1 && msg.sender == player1) {
10         // Make sure that the hashes match and thus player1's commitment was valid.
11
12         // *** BELOW IS THE LINE WITH THE VULNERABILITY ***
13         require(player1Hash == sha256(bytes32(randInt + uint256(choice))), "Hashes 1 don't
14             match!");
15
16         // Store the random integer and choice of player1.
17         player1randInt = randInt;
18         player1Choice = choice;
19         // Update the game state to make sure it's player2's turn to reveal next.
20         gameState = State.NoReveal2;
21
22         // Update the time stamp to check for timeout later
23         timerStart = now;
24     }
25     /**
26     ...
27     */
28 }
29 /**
30 ...
31 */
```

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:

```

1 function revealRock(uint256 randInt) public{
2     // 'revealRock' reveals that the sender choice was Rock, and randInt in the random
    integer he
3     // used to generate the hash.
4     reveal(randInt, RPS.Rock); // RPS.Rock is 0
5 }
6
7 function revealPaper(uint256 randInt) public{
8     // 'revealPaper' reveals that the sender choice was Paper, and randInt in the random
    integer he
9     // used to generate the hash.
10    reveal(randInt, RPS.Paper); // RPS.Paper is 1
11 }
12
13 function revealScissors(uint256 randInt) public{
14     // 'revealScissors' reveals that the sender choice was Scissors, and randInt in the
    random integer he
15     // used to generate the hash.
16     reveal(randInt, RPS.Scissors); // RPS.Scissors is 2
17 }
18
19 function reveal(uint256 randInt, RPS choice) private {
20     require(registered(), "You have not registered for the game!");
21     require(gameInProgress(), "At least one player missing!");
22
23     // player1 always reveals his choice and random integer first.
24     if (gameState == State.NoReveal1 && msg.sender == player1) {
25         // Make sure that the hashes match and thus player1's commitment was valid.
26         require(player1Hash == sha256(bytes32(randInt*10 + uint256(choice))), "Hashes 1 don't
            match!");
27         // Store the random integer and choice of player1.
28         player1randInt = randInt;
29         player1Choice = choice;
30         // Update the game state to make sure it's player2's turn to reveal next.
31         gameState = State.NoReveal2;
32
33         // Update the time stamp to check for timeout later
34         timerStart = now;
35     }
36     /**
37     ...
38     */
39 }

```

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, similarly 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 $260 \bmod 256 = 4$. 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 on 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 $\gcd(10, 2^{256}) \mid l - k$ and therefore, since $\gcd(10, 2^{256}) = 2$ we need l and k to be such that $2 \mid l - k$. 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:

```

1 def xgcd(b, a):
2     x0, x1, y0, y1 = 1, 0, 0, 1
3     while a != 0:
4         q, b, a = b // a, a, b % a
5         x0, x1 = x1, x0 - q * x1
6         y0, y1 = y1, y0 - q * y1
7     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 attack³. 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:

```

1 import hashlib, random
2
3 magic = 92633671389852956338856788006950326282615987732512451231566067206330503711949
4
5 def attack():
6     rand_int = random.randrange(0, magic)
7     rock = magic + rand_int
8     print("rock: {}".format(rock))
9     rock = (rock * 10) % 2**256
10    rock = rock.to_bytes(32, byteorder='big')
11    rock_hash = hashlib.sha256(rock).hexdigest()
12    scissors = rand_int
13    print("scissors: {}".format(scissors))
14    scissors = ((scissors * 10) + 2) % 2**256
15    scissors = scissors.to_bytes(32, byteorder='big')
16    scissors_hash = hashlib.sha256(scissors).hexdigest()
17
18    # as we expect, those two will always be the same
19    print("rock hash: 0x{}".format(str(rock_hash)))
20    print("scis hash: 0x{}".format(str(scissors_hash)))
21
22 attack()

```

2.7.5 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) [2]. 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.

³ $x = 92633671389852956338856788006950326282615987732512451231566067206330503711949$

```

1 function reveal(string nonce, uint choice) public returns (string){
2     /**
3     ...
4     */
5
6     //record time of first reveal
7     if (timeReveal==0) {
8         timeReveal = now;
9     }
10
11    /**
12    ...
13    */
14 }
15
16 function finish() public {
17     //both players need to have revealed or the second player needs to have missed the
        reveal window
18     require ((hasRevealed[player1] && hasRevealed[player2]) || now-timeReveal > 100);
19
20     //avoid re-entrancy
21     uint payout = pot;
22     pot = 0;
23
24     //if one party did not reveal, pay the other party
25     if (hasRevealed[player1] && !hasRevealed[player2]) {
26         player1.transfer(payout);
27         payout=0;
28     }
29
30     if (hasRevealed[player2] && !hasRevealed[player1]) {
31         player2.transfer(payout);
32         payout=0;
33     }
34
35     /**
36     ...
37     */
38 }

```

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 seconds) 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 [3, 4, 5]. 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`” [3], so this would most likely be safe anyway.

3. Game code

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

```

1 pragma solidity ^0.4.16;
2
3 contract rpsContract {
4
5     // This should be self-explanatory
6     struct Player {
7         address add;
8         bool revealed;
9         bool got_paid;
10        uint256 hashed_choice;
11        uint8 choice;
12    }

```

```

13
14     address owner;
15     // Count if both players got paid (in terms of a tie)
16     Player[2] players;
17     // 0 - player 0, 1 - player 1, 2 - tie
18     uint gameWinner;
19     // Used to reset the game in case of inactivity
20     uint256 timer;
21     GamePhase gamePhase;
22     uint256 gameStake;
23
24     // Idle - waiting for new players
25     // Started - one player started the game
26     // Reveal - waiting for the players to reveal
27     // Finished - ready to claim the Ether
28     enum GamePhase { Idle, Started, Reveal, Finished }
29
30     constructor () public {
31         owner = msg.sender;
32         gamePhase = GamePhase.Idle;
33         gameStake = 0;
34         players[0] = Player(0, false, false, 0, 0);
35         players[1] = Player(0, false, false, 0, 0);
36     }
37
38     /**
39     Start a new game or get into an existing one by sending a hashed (sha3)
40     value of the choice and a random (chosen by the player) seed
41     */
42     function play(uint256 hashed_choice) public payable {
43         require (gamePhase == GamePhase.Idle || gamePhase == GamePhase.Started, "Game is
            currently on");
44         require (msg.value >= 1 ether, "Stakes need to be at least 1 ether");
45         require (msg.value % 2 == 0, "Stakes need to be divisible by 2");
46
47         if (gamePhase == GamePhase.Idle) { // first player starting the game
48             // half of what you put in is treated as deposit
49             // and half as the game prize pool
50             gameStake = msg.value / 2;
51             gamePhase = GamePhase.Started;
52             players[0] = Player(msg.sender, false, false, hashed_choice, 0);
53         } else { // second player joining the game
54             require (msg.value / 2 == gameStake, "Stake needs to be equal to the other
                player's stake");
55             gamePhase = GamePhase.Reveal;
56             // start the timer to enable resetting the game
57             // in case of inactivity
58             timer = now;
59             players[1] = Player(msg.sender, false, false, hashed_choice, 0);
60         }
61     }
62
63     /**
64     Reveal your choice by providing the nonce and choice
65     */
66     function reveal(uint256 nonce, uint8 choice) public {
67         require(gamePhase == GamePhase.Reveal, "Game not in reveal phase yet");
68         bytes memory reveal_val = abi.encodePacked(nonce, choice);
69         uint8 p = determinePlayer(msg.sender);
70         require(p != 2, "You are not taking part in the game");
71         require(uint256(keccak256(reveal_val)) == players[p].hashed_choice, "Invalid seed
            and/or choice");
72
73         // save the choice and mark that
74         // the player have revealed the value
75         players[p].choice = choice;
76         players[p].revealed = true;
77

```

```

78     if (players[0].revealed == true && players[1].revealed == true) {
79         // both players revealed
80         // determine winner and allow claiming winnings
81         gamePhase = GamePhase.Finished;
82         gameWinner = getWinner(int8(players[0].choice), int8(players[1].choice));
83     } else {
84         // reset the timer due to activity
85         timer = now;
86     }
87 }
88
89 /**
90 Claim money or reset the game in case of inactivity
91 */
92 function claim() public {
93     if (now - timer > 2 minutes && gamePhase == GamePhase.Reveal) {
94         // prevent reentrancy attack
95         reset();
96
97         // 2 minutes have passed since last activity
98         // allow anyone to reset the game
99         if (players[0].revealed == true) {
100             // first player revealed, gets all
101             players[0].got_paid = true;
102             players[0].add.transfer(4*gameStake);
103         } else if (players[1].revealed == true) {
104             // second player revealed, gets all
105             players[1].got_paid = true;
106             players[1].add.transfer(4*gameStake);
107         } else {
108             // no player revealed, split 50/50
109             players[0].got_paid = true;
110             players[1].got_paid = true;
111             players[0].add.transfer(2*gameStake);
112             players[1].add.transfer(2*gameStake);
113         }
114
115         return;
116     }
117
118     require(gamePhase == GamePhase.Finished, "Game not finished yet");
119     uint8 p = determinePlayer(msg.sender);
120     require(p != 2, "You are not taking part in the game");
121     require(!players[p].got_paid, "You already have your money");
122
123     if (gameWinner == p) {
124         players[p].got_paid = true;
125         // transfer 75% of total to winner
126         msg.sender.transfer(3*gameStake);
127     } else if (gameWinner != p && gameWinner != 2) {
128         players[p].got_paid = true;
129         // transfer 25% of total to loser
130         msg.sender.transfer(gameStake);
131     } else if (gameWinner == 2) {
132         players[p].got_paid = true;
133         // transfer each player 50% of total
134         msg.sender.transfer(2*gameStake);
135     }
136
137     if (players[0].got_paid && players[1].got_paid) {
138         reset();
139     }
140 }
141
142 /**
143 Determine the player based on the address.
144 Returns 2 if player is unknown
145 */

```

```

146 function determinePlayer(address add) private view returns(uint8) {
147     if (players[0].add == add) {
148         return 0;
149     } else if (players[1].add == add) {
150         return 1;
151     } else {
152         return 2;
153     }
154 }
155
156 /**
157 Reset the game state to idle to allow new games
158 to be played
159 */
160 function reset() private {
161     gamePhase = GamePhase.Idle;
162 }
163
164 /**
165 Determine the winner given choices of both players
166 0 + 3k means 'rock'
167 1 + 3k means 'paper'
168 2 + 3k means 'scissors'
169 */
170 function getWinner(int8 a, int8 b) public view returns (uint8) {
171     require(gamePhase == GamePhase.Finished, "Game not finished yet");
172
173     if ((a - b) % 3 == 1) {
174         return 0;
175     } else if ((a - b) % 3 == -1) {
176         return 1;
177     } else if ((a - b) % 3 == 2) {
178         return 1;
179     } else if ((a - b) % 3 == -2) {
180         return 0;
181     } else if ((a - b) % 3 == 0) {
182         return 2;
183     }
184 }
185
186 /**
187 Helper function to be able to see what state
188 the game is currently in
189 */
190 function getGamePhase() public view returns (string) {
191     if (gamePhase == GamePhase.Idle) {
192         return "Waiting for players";
193     } else if (gamePhase == GamePhase.Started) {
194         return "Waiting for player 2";
195     } else if (gamePhase == GamePhase.Reveal) {
196         return "Waiting for the players to reveal their choices";
197     } else if (gamePhase == GamePhase.Finished) {
198         if (gameWinner == 0) {
199             return "Player 1 won, waiting to claim the prize";
200         } else if (gameWinner == 1) {
201             return "Player 2 won, waiting to claim the prize";
202         } else {
203             return "Tie, waiting for the players to claim the money";
204         }
205     }
206 }
207
208 /**
209 Helper function to be able to see what is the
210 stake you need to put in to compete
211 */
212 function getStake() public view returns (uint256) {
213     return gameStake;

```

```
214     }
215 }
```

and the hash generating script in Python

```
1 from web3 import Web3
2 from random import randint
3
4 seed = randint(2**128, 2**256)
5 choice = int(input("Pick a choice: "))
6
7 h = Web3.soliditySha3(['uint256', 'uint8'], [seed, choice%3])
8 print("Seed-choice:", str(seed) + ", " + str(choice%3))
9 print("Hash:", int(h.hex(), 16))
```

References

- [1] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, <https://gavwood.com/paper.pdf>
- [2] Ethereum Block Protocol 2.0, wiki post created by Github users **rucevits** and **heiko-heiko**, <https://github.com/ethereum/wiki/blob/c02254611f218f43cbb07517ca8e5d00fd6d6d75/Block-Protocol-2.0.md>
- [3] Recommendations for Smart Contract Security in Solidity, <https://consensys.github.io/smart-contract-best-practices/recommendations/#timestamp-dependence>
- [4] Geth protocol implementation of the timestamp rule, <https://github.com/ethereum/go-ethereum/blob/4e474c74dc2ac1d26b339c32064d0bac98775e77/consensus/ethash/consensus.go#L45>
- [5] Parity protocol implementation of the timestamp rule, <https://github.com/paritytech/parity-ethereum/blob/73db5dda8c0109bb6bc1392624875078f973be14/ethcore/src/verification/verification.rs#L296-L307>