Blockchain Distributed Ledger

Coursework 2

Madhav Benoi
s1902743

---

1 High Level Design

The contract implements a dice game, where the contract rolls a dice. If it rolls 1-3 **Player A** wins, 4-6 **Player B** wins.
First player to enter the game is **Player A** , the second player to enter is **Player B**. To ensure pseudo randomness a Cryptographic Commitment Scheme is used.

1.1 Enter Game and Commitment stage

The players first join the game by `enter_game(bytes32 userhash)` function, through which they have to enter a commitment.
The players needs to put in at-least 4 ether in-order to player. 3 ether to play the game and 1 ether for other expenses that might incur, gas prices. Most of it will be refunded. Think of this as deposit to fund the gas prices that might happen at certain stage of the game.

A Commitment is done in the form of a `sha256` hash, the format of this hash should be

```
SHA256( address | secret | secret_number )
```

Where,
`address` this is the address of the wallet
`secret` is some `salt` 32 byte string
`secret_number` is the number the player wants to be added to the seed.

Once, both player have joined the game, the game can proceed. If first player wants to back-out (and the 2nd player has not entered the game), they have the option to do so by calling `player_A_back_out()`

Why is the joining and commitment stage the same function?

> In a object oriented programming language, this would be a design that should be desirable, for solidity we need to think about gas efficiency. Designing multiple functions that need to be called in my opinion is a waste of gas, even if gas charges are minute, this adds up to incur heavy costs. We must avoid, unnecessary gas cost when writing solidity.

Can the Player B send the same hash as Player A?

> Yes, they can, but **Player B** would not able to verify their hash as, address of the sender is used in the calculation. Which unless both players have the same address, or there is a hash collision with the input, it is impossible for this to happen.

There are still few flaws in this, which I will discuss in the security part of this.

1.2 Reveal stage

In this stage, the players have to reveal their secret that they used to calculate the hash. They do that by inputting the secret in the `reveal(bytes32 _secret,uint256 _secret_number)` function.
Here even if either of the player wanted to change their input, it's impossible to do as even one byte of change in the input will change the entire hash.
Once, the player reveal their `secret` and `secret_number` the number is used as part of the seed calculation.

What if One of the Player backs out?

> There is `request_timeout()` feature, once a sufficient amount of time has passed since the player's commit time. They have the option to back out of the game and get a refund. The other player can do so as well.
> In the future, they might a penalising mechanism (as there should be consequences for stalling) which punished the other player for stalling, did not implement because of lack of time.

Is the hash secure?

> Well, depends, can it be reversed? Yes theoretically, but the security of hash functions depends on how long it will theoretically take to bruteforce. As, I request a 64 byte secret (32 byte secret nonce and 32 byte secret number), they will need to brute force a 2^{8*64} bits. Which takes almost as long as the age of the universe.

1.3 Game start

Once, both players have revealed the secret we can start the game.
The `secret_numbers` from the players are xor'ed and now in combination with the future blockhash, this is now the seed used for the PRNG.

1.3.1 A little bit about the PRNG

https://en.wikipedia.org/wiki/Permuted_congruential_generator
Permuted Congruential Generator is one of the modern PRNGs used, it's got a order of 2^63 which means PRNG needs to called 2^64 times for the numbers to repeat.
PRNG is used to give a sort of randomness where the output is not obvious with the seed. The reason why I choose to make this function public is for them to verify this function actually gives random numbers from 1-6.

1.3.2 Back to Game Start

Once, the PRNG returns a random number from 1 to 6. We calculate the index by `(number-1)//3` This will return either `0` or `1`. Which indicates `0-2` is `0` or **Player A**, We get the address of winner or loser, we then take money from the loser and then give the winner money they won.
We also give the **Player** who called the function some constant value compensation for gas price. The idea is to split the gas price between the player. I couldn't figure out how to exactly figure out how to get the computation cost, so I just gave a constant value. This could changed in the future.

1.4 Withdraw

The Players can now claim their winnings or loses, this function is made to be reentrancy safe. This function cannot be called by players currently playing the game, because that would introduce a big flaw.

1.5 Miscellaneous Functions

Player A Backing out

This function let's Player A back out of the game if they wanted to

Timeout function

If one of the player does not reveal after 10 minutes have past, this function can be called to end the game and get a refund

Random verify

You can input a seed and verify the output is psuedo random.

1.6 Questions about Design

Who pays for the reward of the winner?*

> The loser of the game pays for the reward of the game, it is only fair as it's not feasible for the contract to pay for winner's winning. The balance of the players in the contract are updated to reflect this.

How is the reward sent to the winner?

> There is `Withdraw()` function, that can be called by anyone as long as they are not playing in the game. The reward has to be withdrawn instead of sent after the game ended as it would introduce a major flaw if you sent it directly from the function.

How is the player guaranteed not to cheat?

> The whole Commitment scheme is one of the methods to guarantee that this is random if both Players play honestly. Ways the players could cheat are avoided :
>
> 1. Knowing what the secret of the other player is before committing, hence picking a number that would be biased towards the adversary winning. This is eliminated by Future blockhash being used. This still doesn't completely eliminate the pseudo randomness.
> 2. Inputting the same hash as the other player is avoided, as the player address is used as one of the fields for the hashing.
>   The commitment scheme basically guarantees that once the player has picked a value and committed, they cannot change their value in the reveal phase. Which is essentially the best source of randomness for the PRNG. Some Security fault will be discussed in later section.

What data type/structures did you use and why?

> Structs for Player : This is a perfect way to encapsulate data related to a Player
> Maps : Hashmaps are very useful way, i used this to store the map address to Struct, this makes the map quite memory heavy but this is vital for mapping player address to their corresponding structs.
> Address array of size 2 for players : this is useful to identify player A and B, it is made constant to save memory and gas. As Vectors (or Extensible arrays in solidity) are not gas efficient or memory efficient.
> uint256 : several places this is used, this data type is critical

> byte32 is used to store the hashes of the commitment, as it is a good way to represent them.

2 Gas Evaluation

2.1 Contract Deployment

Everything is in `units` as gas prices change but the computation remain largely the same in solidity.
Deployed Contract

- Transaction hash :
  `0xcc21f674deb8f20eefed006966e36f5d9b70a4d3eb6175aa430a011d72df8c62`
- Gas : `2233031 units`
- Deployment address : `0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc`
  This is heaviest cost of them, incurred by the owner of the contract. Initialising all the `Struct` and `Hashmaps` is memory heavy, as well as getting all the functions

2.2 Player incurred Gas costs

Enter game

Calling `enter game` by both players on average

- Gas : `120318 units`
  Filling the Structs with values is a heavy operation and having the hash of the function be an argument.
  There is a slight difference in gas units between both players.

Reveal

Calling `reveal` by both players on average

- Gas : `62775 units`
  Nothing much, this is pretty standard and fair between the players.

Start game

Calling `start_game`

- Gas : `133024 units`
  This is going to be a heavy operation so the Player who call this function gets

rewarded some `wei` in compensation and the other Player's balance is subtracted to reflect this.

Withdraw

Calling `Withdraw`

- Gas : `37989 units`
  The biggest issue in my contract in terms of gas efficiency is that there is a lot of type casting from `uint64` to `uint8` or `uint64` to `uint128` this is very intensive for computation, but in-order for me implement my prng step, I need to sacrifice this.

Miscellaneous functions

Timeout

2.3 Improvement techniques

I removed some functions that may be unnecessary and made some things more in-line. I made myself use only one hashmap. I restricted the player array to only size `2` instead of a `Vector`.
I also tried using smaller `uint`s wherever necessary, and also try not to store anything unnecessary.

3 Security Evaluation

Most of the attacks have been explained in the lectures, I will not go over what they mean, I will just say if it is possible or not.

3.1 Minor flaw

Player A has a disadvantage in the first commitment stage as, Player B can look at Player A's hash and could get the value and take it into account for their calculation.

3.2 Choosing not to play flaw

The biggest flaw in the code is that once the either of the Player has revealed their secret in the reveal stage. The other Player can view the evm memory and see what the `random_value` is set to, this can help the second player deduce if they should reveal the number or not, as they can somewhat deduce if it is biased towards theirs or the other players side.
The future blockhash is used to mitigate this, but this can be exploited by a miner to bring the results more favourable to them.

3.3 Griefing

As I am doing a pull over push pattern, I don't think griefing attacks is possible. As I don't use `send()`

3.4 Reentrancy

This attack does not happen in my contract, as I do reset all the values related to the address once the game is completed and the `Withdraw()` function uses the safe `.transfer()` method to send money to the address.

3.5 Front-running

This attack also does not happen in my contract.

3.6 Underflow/Overflows

Whenever there is a underflow/overflow operation, Solidity usually reverts the transaction. I've had some issues with this during the PRNG part, the casting it to bigger numbers were my solution to this, but I am sure there's a more gas efficient way to do it. There might also be overflows if someone puts `2^256 wei` into the contract, but this shouldn't ever happen as the number of `eth` in the network converges to a value lower than this.

4 Trade off

Certainly, gas/computation steps heavy due to wanting to make sure this contract is secure. There are several small details or choices I had to take to
Using the commitment scheme is very heavy on the gas, but this is is vital to keep the game pseudo random.
My contract also isn't very Gas fair, as `Player A` incurs slightly more cost than `Player B` by some order of `10^4` s units.

5 Analysis of fellow student's code

5.1 High level view of my fellow student code

In my fellow student's code, the first step is call the `deposit()` function and deposit some money into the contract, the contract stores the deposit mapping it to the address. Once you've done that, you call the `join` function to join the game. After which you call the `commit()` function to commit a hash, after that you call the `reveal` function to reveal, finally `decide` function which would generate a random number and decide the winner.

Additionally, he had a random bias removal function because the `rngs` he implement was favoured towards one player (due to modulo 6 bias) was very negligible amount ($\frac{4}{2^{64}}$), but it's a waste of gas to have this function, in my opinion. this bias would not even be noticeable. I commend him for thinking about this detail it, but it felt pointless to me.

5.2 Vulnerabilities discovered

There was a vulnerability in my friend's random number generator, in-order for him to eliminate bias,

```
function random(uint seed) public pure returns(uint) {
        uint biasBound = (-6) % 6;
        uint rand = 1;
        uint count = 0;
        while (true) {
            if (rand >= biasBound) {
                break;
            }
            rand = uint(keccak256(abi.encodePacked(seed, count)));
            count += 1;
        }
        return rand % 6;
    }
```

`uint biasBound = (-6) % 6;` is a bug in solidity, as a negative `uint` doesn't do 2s compliment. This would error in solidity or just give `0` (I don't quite remember what the behaviour was).
Some other issues include
`balances[winner] += winnings`
Which paid the winnings in `wei` instead of `ether`.
I also mentioned `Player B` being able to input the same hash in the commitment scheme and just waiting for the other player to reveal.
After pointing these out, my friend did fix the issues.

6 Testing

For anyone, who want to test the code, here's a python script that will encode the player address, secret and number offline, and generates the hash. I used this to generate values for testing.

```python
from hashlib import sha256

def pad(a : bytes):
    return b"\x00"*(32-len(a)) + a

def generate_hash(address : bytes,secret : bytes, number : int  ):
    a = pad(address)
    b = pad(secret)
    c = number.to_bytes(32,"big")
    return sha256(a+b+c).hexdigest()

# ----- Player A ----
address = "DFB302F606c7E8EDE75453DaFdb41862D41F058B"
address = bytes.fromhex(address)
secret = b"this_is_super_secret"
number = 42
print("Player A : ", "0x"+generate_hash(address,secret,number))
print("Secret :" , "0x"+pad(secret).hex())
print("Number :" , number)

# ----- Player B ----
address = "ec432F4595EaD9ab0a374C5fB8E842dAD50BaAA6"
address = bytes.fromhex(address)
secret = b"is_this_super_secret"
number = 0x1337deadbeef
print("Player B : ", "0x" + generate_hash(address,secret,number))
print("Secret :" , "0x"+pad(secret).hex())
print("Number :" , number)
```

7 Transaction history

Contract Creation

```
[block:2798136 txIndex:2] from: 0xDFB...F058B to: Dice.(constructor) value: 0 wei data: 0x608...10033 logs: 0 hash: 0x983...cb59d
status              true Transaction mined and execution succeed
transaction hash    0xcc21f674deb8f20eefed006966e36f5d9b70a4d3eb6175aa430a011d72df8c62
from                0xDFB302F606c7E8EDE75453DaFdb41862D41F058B
to                  Dice.(constructor)
gas                 2233031 gas
transaction cost    2233031 gas
input               0x608...10033
decoded input       {}
decoded output      -
logs                []
val                 0 wei
```

Player A enters the game with 5 ether :

```
[block:2798150 txIndex:0] from: 0xDFB...F058B to: Dice.enter_game(bytes32) 0xCEd...De8Fc value: 5000000000000000000 wei data: 0x5d5...f2fe5 logs: 1 hash: 0x3d6...a214e
status              true Transaction mined and execution succeed
transaction hash    0x40e7211045f9226963e884a0dfeeba5a6b58d78451e289a0ab6ad8728d8e78d8
from                0xDFB302F606c7E8EDE75453DaFdb41862D41F058B
to                  Dice.enter_game(bytes32) 0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc
gas                 109890 gas
transaction cost    109890 gas
input               0x5d5...f2fe5
decoded input       {
                        "bytes32 userhash": "0x7d77c4e7b2af9a41ebb78c493d05768ba4edd077af0e34e0dae8bdc27eff2fe5"
                    }
decoded output      -
logs                [
                        {
                            "from": "0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc",
                            "topic": "0x97aeeadb219a3b268edf774c8c624fb7c29a06801f6976d2110218afc40c84d2",
                            "event": "Player_entered",
                            "args": {
                                "0": "0xDFB302F606c7E8EDE75453DaFdb41862D41F058B",
                                "ad": "0xDFB302F606c7E8EDE75453DaFdb41862D41F058B"
                            }
                        }
                    ]
val                 5000000000000000000 wei
```

Player B enters the game with 5 ether :

```
[block:2798163 txIndex:0] from: 0xec4...BaAA6 to: Dice.enter_game(bytes32) 0xCEd...De8Fc value: 5000000000000000000 wei data: 0x5d5...599c8 logs: 1 hash: 0x90b...72485
status              true Transaction mined and execution succeed
transaction hash    0x82bfe7348c5bd3b20e7479be5f381b0393654dc5f52b4ba9f57fe14108af4462
from                0xec432F4595EaD9ab0a374C5fB8E842dAD50BaAA6
to                  Dice.enter_game(bytes32) 0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc
gas                 130747 gas
transaction cost    130747 gas
input               0x5d5...599c8
decoded input       {
                        "bytes32 userhash": "0x2f25fa97e56086968335356e5fa515fe5586e6c21d1736a21c048c13b3c599c8"
                    }
decoded output      -
logs                [
                        {
                            "from": "0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc",
                            "topic": "0x97aeeadb219a3b268edf774c8c624fb7c29a06801f6976d2110218afc40c84d2",
                            "event": "Player_entered",
                            "args": {
                                "0": "0xec432F4595EaD9ab0a374C5fB8E842dAD50BaAA6",
                                "ad": "0xec432F4595EaD9ab0a374C5fB8E842dAD50BaAA6"
                            }
                        }
                    ]
val                 5000000000000000000 wei
```

Player B reveals their secret :

```
[block:2798179 txIndex:0] from: 0xec4...BaAA6 to: Dice.reveal(bytes32,uint256) 0xCEd...De8Fc value: 0 wei data: 0xa2d...dbeef logs: 0 hash: 0x8fe...d2b42
status              true Transaction mined and execution succeed
transaction hash    0xcb910d19245b874c1ce0258d609da3e8cbb1485f5cb878d73ebfa941491529b8
from                0xec432F4595EaD9ab0a374C5fB8E842dAD50BaAA6
to                  Dice.reveal(bytes32,uint256) 0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc
gas                 70749 gas
transaction cost    70749 gas
input               0xa2d...dbeef
decoded input       {
                        "bytes32 _secret": "0x0000000000000000000000000069735f746869735f73757065725f736563726574",
                        "uint256 _secret_number": "21130680057583"
                    }
decoded output      -
logs                []
val                 0 wei
```

Player A reveals their secret :

```
[block:2798190 txIndex:1] from: 0xDFB...F058B to: Dice.reveal(bytes32,uint256) 0xCEd...De8Fc value: 0 wei data: 0xa2d...0002a logs: 0 hash: 0x186...3146f
status              true Transaction mined and execution succeed
transaction hash    0x1ef30c781059e1deaf38821ed87d35fecc5803683147c1771ff4c1f449eb18fa
from                0xDFB302F606c7E8EDE75453DaFdb41862D41F058B
to                  Dice.reveal(bytes32,uint256) 0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc
gas                 54802 gas
transaction cost    54802 gas
input               0xa2d...0002a
decoded input       {
                        "bytes32 _secret": "0x0000000000000000000000000000746869735f69735f73757065725f736563726574",
                        "uint256 _secret_number": "42"
                    }
decoded output      -
logs                []
val                 0 wei
```

Player A starts the game and we see Player B won been emitted :

[block:2798200 txIndex:0] **from:** 0xDFB...F058B **to:** Dice.start_game() 0xCEd...De8Fc **value:** 0 wei **data:** 0x244...967b3 **logs:** 1 **hash:** 0x1db...30233

status	true Transaction mined and execution succeed
transaction hash	0xb0e80c863660d0201b416c9729151b3879c26b80e2cc1d8e05459fca41626c22
from	0xDFB302F606c7E8EDE75453DaFdb41862D41F058B
to	Dice.start_game() 0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc
gas	133024 gas
transaction cost	66512 gas
input	0x244...967b3
decoded input	{}
decoded output	-
val	0 wei

logs

```
[
	{
		"from": "0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc",
		"topic": "0x2e30273de55771a63c6a3feda73f1f9f8056a9ad1943389579e57a5799527357",
		"event": "Player_winner",
		"args": {
			"0": "0xec432F4595EaD9ab0a374C5fB8E842dAD50BaAA6",
			"ad": "0xec432F4595EaD9ab0a374C5fB8E842dAD50BaAA6"
		}
	}
]
```

Player A withdraws :

[block:2798205 txIndex:0] **from:** 0xDFB...F058B **to:** Dice.Withdraw() 0xCEd...De8Fc **value:** 0 wei **data:** 0x57e...a89b6 **logs:** 0 **hash:** 0xfaf...062a8

status	true Transaction mined and execution succeed
transaction hash	0xc1a1526d5f2d5831101643071f1645ca50a661389fb02227346a79651e45acd6
from	0xDFB302F606c7E8EDE75453DaFdb41862D41F058B
to	Dice.Withdraw() 0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc
gas	37989 gas
transaction cost	20736 gas
input	0x57e...a89b6
decoded input	{}
decoded output	-
logs	[]
val	0 wei

Player B Withdraws :

[block:2798208 txIndex:1] **from:** 0xec4...BaAA6 **to:** Dice.Withdraw() 0xCEd...De8Fc **value:** 0 wei **data:** 0x57e...a89b6 **logs:** 0 **hash:** 0x860...095db

status	true Transaction mined and execution succeed
transaction hash	0x20b7bbf816d9ece97beee7572ff73424851220d2c815024018c67c5e000e53af
from	0xec432F4595EaD9ab0a374C5fB8E842dAD50BaAA6
to	Dice.Withdraw() 0xCEd315B49B2084B78391bfe0fa1AB29Fe04De8Fc
gas	37989 gas
transaction cost	20736 gas
input	0x57e...a89b6
decoded input	{}
decoded output	-
logs	[]
val	0 wei

8 Code

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

contract Dice {
    // players and owner info
    address owner;
    address [2] public players;

    // 10 minutes in Unix eppoch
    uint256 DECAY = 300000 * 2;

    // When both players have entered the game, this will be set to True
    bool game_started = false;

    // Takes the lucky numbers of the Users to calculate the seed,
player_A_value ^ player_B_value ^ latest_block_hash
    uint256 random_value = 0;

    /*
        Stores information of each Players current and older
        solutionHash : Hash committed by the player
        commitTime : Timestamp when the block for the transaction was mined
        deposit : How much the player has deposited to play the game
        revealed : has the Player revealed the secret to get their hash?

    */
    struct Player {
        bytes32 solutionHash;
        uint commitTime;
        uint256 deposit;
        bool revealed;
    }

    // This maps the address of the player to their corresponding structs
    mapping(address => Player) public comm;

    // prng constants
https://en.wikipedia.org/wiki/Permuted_congruential_generator
    uint64 state = 0x4d595df4d0f33173;
    uint64 multiplier = 6364136223846793005;
    uint64 increment = 1442695040888963407;

    // events
```

```
    event Player_entered(address ad);
    event Player_winner(address ad);

    // Ensures that the contract know who the owner is once the contract is
created
    constructor() {
        owner = msg.sender;
    }

    // ----------------- Game Functions -----------------------

    // Order in which the Player must player
    // enter_game -> reveal -> start_game

    /*
     This is how players first enter the game, they have to deposit 4 ether
to play, (3 to play and 1 for any gas prices incurred)

     User has to input a hash to play the game
   */
    function enter_game(bytes32 userhash) public payable {
        require(msg.value >= 4 ether, "Need 4 ether to play the game");
        require(!game_started,"A game is already in session");
        require(comm[msg.sender].commitTime == 0, "You have already
Commited");

        // Add players to the array, if player A has joined, we add Player B
to array
        if (players[0]==address(0)){
        players[0] = payable(msg.sender);
        }
        else {
            players[1] = payable(msg.sender);
            game_started = true;

            }
        // Set the struct of the player
        comm[msg.sender].commitTime = block.timestamp;
        comm[msg.sender].solutionHash = userhash;
        comm[msg.sender].revealed = false;
        comm[msg.sender].deposit += msg.value;
        emit Player_entered(msg.sender);

    }



    /**
```

```
    /**
        Reveal phase of Commitment Scheme
        Ask for a  Secret nonce and a secret number, this is now hashed and
compared to the preprocessed hash
        H( address | _secret  | _secret_number)
        where, H() is the sha256 function
        Security : This is the same as bruteforcing a 64 byte secret, which
takes more than the age of the universe to compute

    **/
    function reveal(bytes32 _secret,uint256 _secret_number) public
onlyPlayers_and_game_start {
        // make sure someone can't reveal again
        require(!comm[msg.sender].revealed, "You have already revealed your
answer");
        // encode the input before hashing it
        bytes memory input = abi.encode(msg.sender,_secret,_secret_number);
        bytes32 solutionHash = sha256(input);
        // Important step to make sure the hashes match
        require(solutionHash==comm[msg.sender].solutionHash,"Hashes do not
match! Try again!");
        // Once revealed, part of the seed is constructed
        comm[msg.sender].revealed = true;
        random_value ^= _secret_number;
        }



    /**
      Game start, steps innolve
      1) Generate the seed and get random number
      2) Get the winner and loser through the index
      3) Calculate the money won or lost
      4) Recalculate the balances
      5) Reset the game to be played again
    **/
    function start_game() public onlyPlayers_and_game_start {
        // Make sure both players have revealed their commitment
        require(comm[players[0]].revealed && comm[players[1]].revealed, "One
of the players have not revealed!");

        // We use the latest blockhash as an additional value for the seed,
        uint256 blockHashNow = uint256(blockhash(block.number-1));
        uint64 seed = uint64( (random_value^blockHashNow));
        // Use our PRNG to generate a number from (1 - 6)
        uint8 lucky_number =  random(seed);
        uint8 index = (lucky_number-1)/3; // ranges from 0 - 5, 0-2 returns
```

```
0, 3-5 returns 1 as there is no floating point

        // Get the winner and loser addresses
        address winner = players[index];
        address loser =  players[(index+1)%players.length];

        // Calculate ether that will be won or lost
        uint256 money_won_or_lost = ((lucky_number-1)%3 + 1)* ( 1 ether);
        comm[winner].deposit += money_won_or_lost;
        comm[winner].deposit -= 70 * 10000 gwei; // charge the player for
gas
        comm[loser].deposit -= money_won_or_lost;
        comm[loser].deposit -= 70 * 10000 gwei; // charge player for gas

        // the caller of this functions get the gas + small reward
        comm[msg.sender].deposit += 140 * 10000 gwei;

        // restart and log winner
        restart();
        emit Player_winner(winner);
    }

    /*
    ----------- Other functions ----------
        Withdrawing money after a game
        Backing out if you don't wish to play
        Timeout if the other player

    */

    /*
    Withdraw your funds from the contract
    Safe way to withdraw funds avoiding Reintrancy bugs
    */
    function Withdraw() public {
        require(players[0] != msg.sender || players[1] != msg.sender,
"Cannot withdraw if you are one of the player, back out first!");

        uint256 b = comm[msg.sender].deposit;
        comm[msg.sender].deposit = 0;
        payable(msg.sender).transfer(b);
    }

    // if the first player wants to back out and no longer wants to play the
game
    function player_A_back_out() public{
```

```
        require(!game_started, "Both player's have enter the game, cannot
backout unless there is a timedout");
        require(players[0]== msg.sender);
        reset(msg.sender);
        players[0] = address(0);
    }

    /*
     Request a Timeout if the game isn't over even after 10 minutes have
passed
    */
    function request_timeout() public onlyPlayers_and_game_start {
        require(comm[msg.sender].commitTime + DECAY  < block.timestamp );
        restart();
    }
    /*
    If both players are stalling and the game can't move on, the owner can
intervene and restart the game
    Potentially have it open to everyone on the chain, but I am unsure how
to prevent it from DDosing the contract
    */
    function owner_timeout() public onlyOwner{
        restart();
    }

    // helper function to restart the game
    function restart() private {
        reset(players[0]);
        reset(players[1]);
        players[0] = address(0);
        players[1] = address(0);
        game_started = false;
        random_value = 0;
    }

    // helper function to reset the Player info struct except their deposit
    function reset(address a) private{
        comm[a].solutionHash = bytes32(0);
        comm[a].commitTime = 0;
        comm[a].revealed = false;
    }

    // ----------------- Psuedo Random Number generator --------------------
---

    // prng info :
```

```
https://en.wikipedia.org/wiki/Permuted_congruential_generator

    // Bitwise circular rotation of a 32 bit number
    function rotr32(uint32 x, uint8 r) private pure returns(uint32) {
        uint32 y =  uint32((x << ((32-r) & 31)));
        return x >> r | y;
    }

    // Main part of the PCG
    function pcg64() private returns(uint32) {
        uint64 x = state;
        uint8 count = uint8(state>>59);
        // due to the nature of multiplication we need to do this to ensure
there is no revertion
        state = uint64(uint(x) * uint(multiplier) + uint(increment));
        x ^= x>>18;
        return rotr32(uint32(x>>27),count);
    }

    function random(uint64 seed) public  returns(uint8)  {
        state = seed+increment;
        /*
        This is a common technique in prngs to ensure unpredictable
randomness
        */
        for(uint16 i=0; i<2; i++){
        pcg64();
        }
        return uint8 (pcg64()%6 + 1);
    }
    // ---------------------- Getters ---------------------------


    function getPlayer() public view returns(string memory){
        require(players[0] == msg.sender || players[1] == msg.sender, "You
are not playing the game");
        if (players[0]==msg.sender){
            return "Hello, Player A!";
        }
        else {
            return "Hello, Player B!";
        }
    }




    // ---------------------- Modifi...
```

```
    // ---------------------- Modifiers ---------------------------

    modifier onlyOwner {
    require(msg.sender == owner);
    _;
    }
    modifier onlyPlayers_and_game_start {
    require(msg.sender == players[0] || msg.sender == players[1]);
    require(game_started);
    _;
    }

}
```