University of Edinburgh

Assignment #2

Student name: *Jure Sternad (s2450797)*

Course: *Blockchains & Distributed Ledgers*
Due date: *October 31, 2022*

1. Description of the contract

For the functionality of the contract, I have decided to involve the availability of using multiple parallel games at once, although it does somewhat increase the cost of gas. Commit and reveal scheme is used for producing the random number.

I declared two structs `Player` and `Roll`. Struct `Roll` is used for storing all values regarding the specific roll while struct `Player` stores hash of the value, value and prize that player can later withdraw. There is one main mapping `rolls` and one that is inside of the struct of the round, `players`.

First player can start the specific roll by calling `commitRoll` function in which he adds Id of the roll and hash of the value. After both players commit their hashes, they can reveal their values. They do that by calling `revealRoll` function in which they add value, salt they used and Id of the roll. The function than checks the matching of previously submitted hashes and in positive result stores the value. After both (or in some cases just one) reveal their vote, any of them can finish the roll and compute the random number `n`. Random number is computed as `xor` of both revealed values. After roll is finished, the players can withdraw their prizes.

2. Reward and cheating

Both players at first commit their hashes by calling the function `commitRoll`. To lock their ether in the contract, they need to add the value of transaction of 3 ether. After the roll finishes, both players can withdraw the amount that they are entitled to. For example, if `n = 2`, first players wins a prize of 2 ether and withdraws 5 ethers while second withdraws 1 remaining ether.

One tactic that player could use would be waiting for other player to reveal his/her value before committing his own. He could then compute `n` on his own and in the case that he would lose, most likely would not call the function to reveal as it would only lead to more gas loss. That is prevented by two `uint256` parameters: `finishStartDate` and `finishEndDate`. The parameters are declared automatically after the second player commits. `finishStartDate` starts 2 minutes after he commits and `finishEndDate` starts 2 minutes after that (the timings in real contract would be much longer). One of the reasons for parameters being declared after

the second player commits is that in other case where they would be declared after the first commits, the second player could wait with his commit and reveal until the `finishStartDate` would almost start. That could prevent first player from revealing in time. He would then be able to withdraw all of the ether (6).
Furthermore, any smart player could use that tactic in order to prevent himself from revealing without any gain and only gas loss.

Before going further in that scenario, let us look at another parameter used for security. Parameter `active` in struct roll is used for securing many of the possible vulnerabilities. It prevents anyone from trying to disrupt the specific roll. Before a specific roll is produced by calling the `commitRoll` function, value of the active is initially 0. For each commit (out of two) of the roll, parameter active is increased by 1. When players reveal their votes, the function `revealRoll` also increases active by 1 for each reveal.

When any of the players wants to finish the game, he/she calls the `finishRoll` function. It is more likely that the player who won will first call that function as he might have higher motivation for withdrawing his ether as soon as possible. The function can only be called in a duration of 2 minutes: after `finishStartDate` and before `finishEndDate`. To call the function, at least one of the players must have revealed his value. That is insured by checking the `active` parameter of the roll which must be either 3 or 4.

In the first case, when `active` is 4 (both players have revealed), the function computes the random value with help of another function `xorRandom` which returns xor of given values. Function later stores the computed prizes in the struct `Player`, mapped from the struct `Roll`.

The second case, in which `active` is 3 (only one player has revealed), the function checks if caller really is the only player who has revealed, and that no one else has revealed. Without any computation of n, it then stores the prize of that player to 6.

In both cases the `active` parameter increases, in first for 1, in second for 2, so in the end it is 5 in either case. That prevents any double use of `finishRoll` function and prepares the prizes to withdraw. For withdraw, **pull over push** is used with `transfer` function. Although `call` which is now more prefererd, could be used, in order to provide more security and further prevent reentrancy, I have decided to use `transfer`.

Another thing that provides more security and prevents either player from cheating is the use of `keccak256` hash function. Both players concatenate their value, another value (salt) and their Etherum address. Involving their address into the hash results in ensuring that the hash was really produced by that player while salt is just added for some more security in order if someone would guess the hash and remove the address from an unhashed message, he would not be able to split the

value and salt.

Another scenario would be that one of the players would commit and other would decide not to. If the player who has committed would not be able to find a new partner that would unfortunately result in some lost ether. The player could then play a game by himself and pay himself for the gas of both commits, although he could avoid spending gas for second reveal and would just wait till the game goes to finish stage.

Finally to mention in this report that takes care of more security is the prevention of double committing and double revealing. Both are implemented in the functions by `require` statement.

3. Data structures and types

As mentioned, two structs are declared. First struct `Roll` is consisted of:

- `address[] revealedPlayers`- used to store players after they successfully reveal their values. It is later used in the finishing stage to use their address in order to get values from mapping `players`;
- `mapping(address => Player) players`- used to map players with their submitted hashes and values, and prizes;
- `uint256 active, uint256 finishStartDate and uint256 finishEndDate`- previously explained.

Struct `Player`:

- `bytes32 valueHash` - hash the player has commited;
- `bytes32 valueHash` - value the player has revealed;
- `bytes32 valueHash` - prize (+/- deposit).

4. Gas

4.1. Gas costs. For a better understanding of costs, I have used hardhat-gas-reporter. The deployment of the contract on average costs 898554 wei which is currently priced at 3340,32 pounds. On average, the commit function costs 84323 wei (313,46 pounds), reveal function 75234 (279,67 pounds), finish function 94924 wei (352,87 pounds) and withdraw function 34272 wei (127,40 pounds). More detailes can be seen in the picture.

Currently, completing one game would on average cost both players 193829 (720,54 pounds) while one of them would need to be additionally charged with 94924 wei (352,87 pounds) worth of gas.

Solc version: 0.8.7	Optimizer enabled: true	Runs: 1000	Block limit: 30000000 gas
Methods			
Contract	**Method**	**Min**	**Max**	**Avg**	**# calls**	**usd (avg)**
s2450797	commitRoll	73406	96800	84323	30	–
s2450797	finishRoll	59979	89801	75234	13	–
s2450797	revealRoll	83914	100852	94924	20	–
s2450797	withdraw	30243	34943	34272	14	–
Deployments					**% of limit**	
s2450797		–	–	898554	3 %	–

4.2. Gas fairness. I tried to make my contract as gas fair as possible. Although the player that first commits (and starts a new game) does need to pay more gas in order to store a new struct on the blockchain, that is to some extent compromised with the second player being charged for setting the finish period start and end date. Reveal function is similarly gas fair to both of the players. The function for finishing the roll is in terms of gas fairness not fair equally to both as it can only be called by one of the players. However, neither of the players is forced to call the function. Any of them can wait for the other to call it although it would be most likely that the player who won the prize would have higher motivation to call it.

Pull over push approach also provides more gas fairness as it does ensure that each of the players pays their own costs of withdrawal.

4.3. Gas efficiency.

- Although many of the integer variables could be stored as smaller, for example `prize` and `active` could be `uint8` since they cannot exceed 6; `value` could also be smaller and still provide enough range for randomness; `finishStartDate` and `finishEndDate` could be `uint48` or even `uint32` (could work only until somewhere after 2100); they all are declared as `uint256` because all of them involve being used for computation. Because the EVM would need to transform them into `uint256` every time they would be computed, that would result in more gas cost for players. However, storing bigger values also results in more gas cost but in my opinion the used approach is more gas efficient and fair as most of the variables are involved in several computations.

5. Potential hazards and vulnerabilities

- Another contribute to gas efficiency is also the use of `external` instead of `public` visibility of the functions.

- Solidity compiler optimizer is also turned in order to produce even more optimized bytecode.
- Mappings are used insted of arrays as they cost less gas.
- Other gas efficiency techniques are described throughout the report.

Found vulnerabilities:

- One of the vulnerabilities that I have found in my contract is the use of `block.timestamp` which is used for comparing the current time with timings timings of the finish stage. As miners can manipulate that, it could result in some attacks. However, I still think that the motivation to perform such an attack is too low based on the prize.
- Another vulnerability is the use of a `transfer` function instead of `call`. Although transfer function could fail in the case that it sent the ether to a contract account which rejects the payment and triggers an event, it sends only 2300 gas which is not enough to perform an attack with a complex code. `call` on the other hand can transfer all of the gas to the receiving contract. This could result in executing complex operations which the caller would pay for. Function `call` is also more vulnerable to reentrancy attacks.
- The last but very important vulnerability that I have found is a case where an attacker could somehow perform an attack which would drain the balance of the contract. That would result in attacking all of the games which are still in progress. That is a vulnerability of providing the ability to play multiple games at once.
- Some other vulnerabilities and how are they mitigated are already described throughout the paper.

6. Tradeoffs

- The first main tradeoff between gas fairness and efficiency is the avaliability of multiple parallel games. Although this does result in more gas cost, it can allow more users to the play game without having to wait for the previous game to end.
- A big tradeoff between security and performance is the use of commit-reveal scheme. The scheme is used for producing the random number n. It, therefore, forces the user into using more functions: one for commit, one for reveal

and one to finish. Although random variable could be generated by block information such as hash or timestamp, that would not be secure as miner could manipulate.

- Another important tradeoff between security and performance is the use of pull over push approach. Although it could be avoided and would be possible to transfer ether by calling the `finishRound` function, as mentioned, that could lead to attacks such as reentrancy.
- Worth mentioning is also another tradeoff between security and performance which is the use of keccak256 function with the value, salt and address of the player. Although it forces the user to compute the hash on his own, it does provide much security, as mentioned before.
- Another tradeoff between security and performance is the time restrictment of the finish stage. Players do need to wait sometime in order to finish the roll and withdraw, and they are limited to the time they have to reveal. However, that ensures security on many levels and helps solve the scenario when one of the players decides to not reveal.
- A tradeoff between efficiency and security is also the fact that deposits of all games are kept together and not stored in a specific game's struct. This potentially could let an attacker who can attack the contract by draining ether to steal ether of the whole contract, resulting in ether being stolen from all of the open games.
- The last important tradeoff is the tradeoff between fairness and efficiency. The `finishRoll` function, which one of the players must sacrifice to call ensures that n can be computed only once and prevents the scenario where both players would need to pay additional gas in order to withdraw their prize.

7. Analysis of a fellow student's contract

I have analysed the contract `s2457006.sol`. The following vulnerabilities were found:

- The most concerning issue is the possible reentrancy attack on several functions. Two of these are `withdrawPlayerA` and `withdrawPlayerb`. As these two functions are almost identical and differ only by the player who can call a function, describing one function should satisfy describing the other. In the code snippet it can be seen that state changes are not properly finished before calling function `transfer`.

```
function withdrawPlayerA() public payable{
    require(playerA[0] == msg.sender, "You are not player A.");
    require(playerA.length == 1, "There is no player.");
    uint256 withdraw = 2 ether;
    payable(playerA[0]).transfer(withdraw);
    playerA = new address payable[](0);
}

function withdrawPlayerB() public payable{
    require(playerB[0] == msg.sender, "You are not player B.");
    require(playerB.length == 1, "There is no player.");
    uint256 withdraw = 2 ether;
    payable(playerB[0]).transfer(withdraw);
    playerB = new address payable[](0);
}
```

An attacker could perform a reentrancy attack by declaring his address as a contract which would when receiving ether call a fallback function `receive` which could later call the game contract (`s2457006.sol`) and use the function `withdrawPlayerA` again. That could be performed as long as the contract has ether.

```
receive() payable external {

    if (s2457006.balance >= msg.value) {
        s2457006.withdrawPlayerA();
    }
}
```

Another similar reentrancy attack could be performed on the `Play` function. A reentrancy attack which would involve mix of those two functions could also be executed. For example, it could call the function `withdrawPlayerA` when the function `Play` would transfer ether to the attacking contract. The issue could be resolved by using the pull over push approach.

```
function Play() public payable {
    require(playerA.length + playerB.length == 2, "There's no enough player.");
    uint256 dice = DiceResult() * 1 ether;

    // Give back the initial 3 ether to the winner.
    uint256 trf = dice + 3 ether;

    if (dice < 4) {
        payable(playerA[0]).transfer(trf);

        // Transfer all the remaining eth to the loser.
        payable(playerB[0]).transfer(address(this).balance);
    }
    else {
        payable(playerB[0]).transfer(dice - 3);

        // Transfer all the remaining eth to the loser.
        payable(playerA[0]).transfer(address(this).balance);
        }

    emit Result(dice);
    emit Players(playerA[0], playerA[0].balance);
    emit Players(playerB[0], playerB[0].balance);

    // Reset the players so that new players have to pay to play this game.
    playerA = new address payable[](0);
    playerB = new address payable[](0);
}
```

- Another issue is the generation of random number which uses `block.timestamp` which a miner could manipulate.

```
function getRandomNumber() public view returns (uint) {
    return uint(keccak256(abi.encodePacked(owner, block.timestamp)));
}

function DiceResult() public view returns (uint256) {
    // Generate random number for 0 to 5
    uint256 index = getRandomNumber() % 6;
    return index + 1;
}
```

- The third issue is the declaration of pragma version. It is best practice to include only one version of pragma which should be stable and recommended by other developers and security professionals.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;
```

- A minor issue is the use of visibility of `public` functions which could be `external`. It is best practice in terms of optimization, to use `external` visibility for functions that are not called from the contract.

8. Transaction history of an execution of a game on my contract

This is included in a separate file `s2450797_history.json`.

9. Tests and coverage

I have also tested my contract and used coverage. Tests are also provided in the separate file `s2450797.ts`. Here are the tests and coverage reports:

```
Successfully generated 6 typings!
Compiled 1 Solidity file successfully

Network Info
============
> HardhatEVM: v2.12.0
> network:    hardhat



  s2450797.sol
    Committing
      ✔ Should let player to commit hash (162ms)
      ✔ Shouldn't let player to commit hash if value != 3 (40ms)
      ✔ Shouldn't let player to double commit
      ✔ Shouldn't let third player to commit (41ms)
    Revealing
      ✔ Should let both players to reveal (82ms)
      ✔ Shouldn't let player to double reveal (49ms)
      ✔ Shouldn't let third player to reveal (54ms)
    Finishing
      ✔ Should let both players to reveal and one to finish (59ms)
      ✔ Should let player to finish in case only he/she revealed (50ms)
      ✔ Shouldn't let player to finish in case no one revealed
      ✔ Shouldn't let player to finish after roll has been already finished (51ms)
      ✔ Shouldn't let player to finish before/after finish period (51ms)
      ✔ Shouldn't let player to finish in case he/she didn't reveal and other did (43ms)
    Game
      ✔ Game 1  (77ms)
      ✔ Game 2  (76ms)
      ✔ Game 3  (71ms)
      ✔ Game 4  (57ms)


  17 passing (1s)

---------------|----------|----------|----------|----------|----------------|
File           |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
---------------|----------|----------|----------|----------|----------------|
 contracts/    |      100 |    88.89 |      100 |      100 |                |
  s2450797.sol |      100 |    88.89 |      100 |      100 |                |
---------------|----------|----------|----------|----------|----------------|
All files      |      100 |    88.89 |      100 |      100 |                |
---------------|----------|----------|----------|----------|----------------|
```

10. Code of my contract

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.7;


contract s2450797 {

    //=====================================================================
    // Errors
    //=====================================================================

    string internal constant ERR_COMMIT_NOT_ACTIVE = "Both players have already committed";
    string internal constant ERR_REVEAL_NOT_ACTIVE = "Reveal phase is not active";
    string internal constant ERR_ALREADY_COMMITED = "You have already committed";
    string internal constant ERR_ALREADY_REVEALED = "You have already revealed";
    string internal constant ERR_NO_MATCH_TRANSACTION_VALUE = "The value of the transaction is not 3 ETH";
    string internal constant ERR_NO_MATCH_HASH = "Hash does not match";
    string internal constant ERR_NOT_FINNISHED = "Finish period is not active";
    string internal constant ERR_NOT_ELIGIBLE = "You are not eligible to finish the roll";
    string internal constant ERR_NO_WITHDRAW = "Withdraw for the roll is not yet available";
    string internal constant ERR_NOT_REVEALED = "None of the players have revealed their value";
    string internal constant ERR_ALREADY_FINISHED = "Roll has been already finished";

    //=====================================================================
    // Data Structures
    //=====================================================================

    struct Player {
        bytes32 valueHash;
        uint256 value;
        uint256 prize;
    }

    struct Roll{
        address[] revealedPlayers;
        mapping(address => Player) players;
        uint256 active;
        uint256 finishStartDate;
        uint256 finishEndDate;
    }

   mapping(uint256 => Roll) public rolls;
```

```
    //=====================================================================
    // Events
    //=====================================================================

    event Commited(address _player, uint256 _id, uint256 _finnishStartDate, uint256 _finnishEndDate);
    event Revealed(address _player, uint256 _id, uint256 _value);
    event Finished(address _player, uint256 _id);
    event Withdrawn(address _player, uint256 _id, uint256 _price);

    //=====================================================================
    // Functions
    //=====================================================================

    /**
     * @notice Committs player's hash
     * @param _valueHash                Hash of the player's value combined with salt and his address
     * @param _rollId                   Id of the roll
     */
    function commitRoll(bytes32 _valueHash, uint256 _rollId) external payable {
        require(rolls[_rollId].active <= 1, ERR_COMMIT_NOT_ACTIVE);
        require(rolls[_rollId].players[msg.sender].valueHash == bytes32(0), ERR_ALREADY_COMMITED);
        require(msg.value == 3 ether, ERR_NO_MATCH_TRANSACTION_VALUE);
        rolls[_rollId].players[msg.sender].valueHash = _valueHash;
        if(rolls[_rollId].active == 1) {
        rolls[_rollId].finishStartDate = block.timestamp + 1 minutes; //this would be bigger in the real contract
        rolls[_rollId].finishEndDate = block.timestamp + 2 minutes;  //this would be bigger in the real contract
        }
        rolls[_rollId].active += 1;
        emit Commited(msg.sender, _rollId, rolls[_rollId].finishStartDate, rolls[_rollId].finishEndDate);
    }

    /**
     * @notice Reveals player's hash
     * @param _value                    Player's value
     * @param _salt                     Salt he used
     * @param _rollId                   Id of the roll
     */
    function revealRoll(uint256 _value, uint256 _salt, uint256 _rollId) external {
        require(block.timestamp <= rolls[_rollId].finishStartDate , ERR_REVEAL_NOT_ACTIVE);
        require(rolls[_rollId].active == 2 || rolls[_rollId].active == 3, ERR_REVEAL_NOT_ACTIVE);
        require(rolls[_rollId].players[msg.sender].value == 0, ERR_ALREADY_REVEALED);
        bytes32 hashedValue = rolls[_rollId].players[msg.sender].valueHash;
        require(keccak256(abi.encodePacked(_value, _salt, msg.sender)) == hashedValue, ERR_NO_MATCH_HASH);
        rolls[_rollId].players[msg.sender]= Player (0,_value, 0);
        rolls[_rollId].revealedPlayers.push(msg.sender);
        rolls[_rollId].active += 1;
        emit Revealed(msg.sender, _rollId, _value);
    }
```

```
/**
 * @notice Finishes the roll (sets the prizes)
 * @param _rollId                   Id of the roll
 */
function finishRoll(uint256 _rollId) external {
    require(
        block.timestamp >= rolls[_rollId].finishStartDate &&
        block.timestamp <= rolls[_rollId].finishEndDate,
        ERR_NOT_FINNISHED
        );
    require(rolls[_rollId].active == 3 || rolls[_rollId].active == 4, ERR_ALREADY_FINISHED);
    if(rolls[_rollId].active == 4) { //case when both players have revealed
        address[] memory _revealedPlayers = rolls[_rollId].revealedPlayers;
        uint256 n = xorRandom(
            rolls[_rollId].players[_revealedPlayers[0]].value,
            rolls[_rollId].players[_revealedPlayers[1]].value
        );
        n += 1;
        if(n < 4) {
           rolls[_rollId].players[_revealedPlayers[0]].prize = 3 + n;
           rolls[_rollId].players[_revealedPlayers[1]].prize = 3 - n;
        }
        else {
           rolls[_rollId].players[_revealedPlayers[1]].prize = n;
           rolls[_rollId].players[_revealedPlayers[0]].prize = 6 - n;
        }
        rolls[_rollId].active += 1;
    }
    if(rolls[_rollId].active == 3) { //case when only one of the players has revealed
        require(
            rolls[_rollId].revealedPlayers[0] == msg.sender &&
            rolls[_rollId].revealedPlayers.length == 1,
            ERR_NOT_ELIGIBLE
            );
        rolls[_rollId].players[msg.sender].prize = 6;
        rolls[_rollId].active += 2;
    }
    emit Finished(msg.sender, _rollId);
}

/**
 * @notice Withdraws player's ether
 * @param _rollId                  Id of the roll
 */
function withdraw(uint256 _rollId) external {
    require(block.timestamp >= rolls[_rollId].finishEndDate, ERR_NO_WITHDRAW);
    require(rolls[_rollId].active == 5, ERR_NOT_REVEALED);
    uint256 withdrawal = rolls[_rollId].players[msg.sender].prize;
    rolls[_rollId].players[msg.sender].prize = 0;
    emit Withdrawn(msg.sender, _rollId, withdrawal);
    payable(msg.sender).transfer(withdrawal*(10**18));
}
```

```
    /**
     * @notice Computes xor of a and b, returns the result
     */
    function xorRandom(uint256 a, uint256 b) public pure returns(uint256) {
        uint256 n = (a ^ b) % 6;
        return n;
    }
}
```